

(12)

(21) 2 319 600

(51) Int. Cl. 7: G07F 17/32

(22) 14.09.2000

(30) 09/396,190 US 14.09.1999

(71) INNOVATIVE GAMING CORPORATION OF  
AMERICA,  
4750 Turbo Circle, RENO, XX (US).

(72) BENDALL, ERIC (US).  
JOHNSON, PETER J. (US).

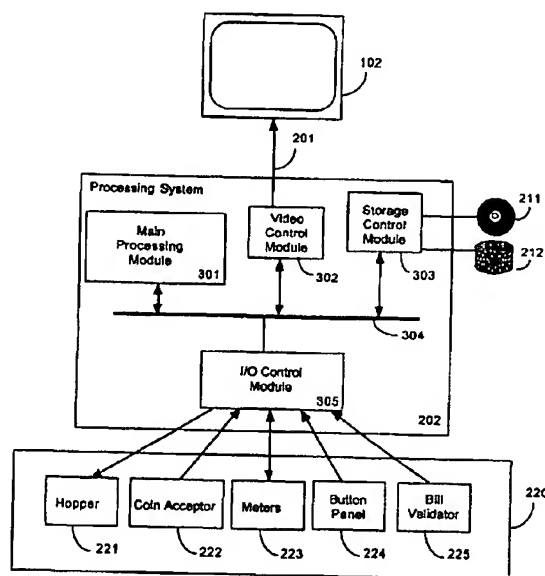
(74) OYEN WIGGS GREEN & MUTALA

(54) METHODE ET APPAREIL FOURNISSANT UNE ARCHITECTURE DE PARTITIONNEMENT  
D'INSTRUCTIONS DE JEU DANS UN APPAREIL DE JEUX DE HASARD

(54) METHOD AND APPARATUS FOR PROVIDING A COMPARTMENTALIZED GAME INSTRUCTION  
ARCHITECTURE WITHIN A GAMING MACHINE

(57)

An apparatus for compartmentalizing various game instruction modules within an overall architecture of a video gaming machine. The video gaming machine comprises a video display, an audio output module, and a plurality of user input buttons to provide reusable instruction modules common to all games implemented on a video gaming platform. The apparatus comprises an open-source operating system, a plurality of platform support and utility modules common to all games run on the video gaming machine, and a plurality of game specific modules and associated data files, the plurality of game specific modules and associated data files comprise a game specific engine module for executing the instructions implementing the specific game on the video gaming machine. The plurality of platform support and utility modules comprise a wagering funds module, a user interaction module, a multimedia output module, and a random number generator module. The plurality of game specific modules and associated data files, the plurality of game specific modules and associated data files comprise a game specific engine module for executing the instructions implementing the specific game on the video gaming machine, a plurality of video image data files combined by the game specific modules to create a sequence of image displayed on the video display; and a plurality of audio sound data files played by the game specific modules to create a series of sounds generated on the audio output module in conjunction with the sequence of image displayed on the video display.





(72) JOHNSON, PETER J., US

(72) BENDALL, ERIC, US

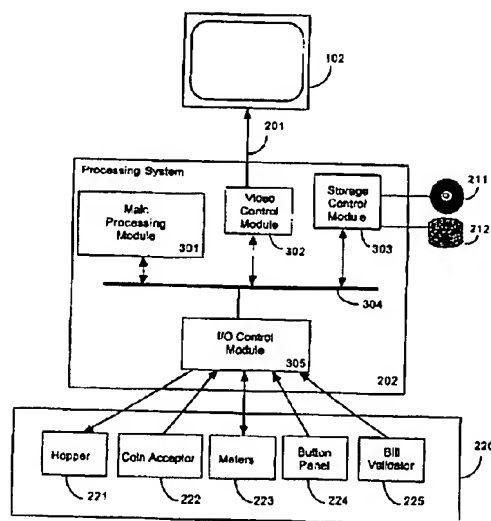
(71) INNOVATIVE GAMING CORPORATION OF AMERICA, US

(51) Int. Cl. <sup>7</sup> G07F 17/32

(30) 1999/09/14 (09/396,190) US

(54) METHODE ET APPAREIL FOURNISSANT UNE  
ARCHITECTURE DE PARTITIONNEMENT D'INSTRUCTIONS  
DE JEU DANS UN APPAREIL DE JEUX DE HASARD

(54) METHOD AND APPARATUS FOR PROVIDING A  
COMPARTMENTALIZED GAME INSTRUCTION  
ARCHITECTURE WITHIN A GAMING MACHINE



(57) An apparatus for compartmentalizing various game instruction modules within an overall architecture of a video gaming machine. The video gaming machine comprises a video display, an audio output module, and a plurality of user input buttons to provide re-usable instruction modules common to all games implemented on a video gaming platform. The apparatus comprises an open-source operating system, a plurality of platform support and utility modules common to all games run on the video gaming machine, and a plurality of game specific modules and associated data files, the plurality of game specific modules and associated data files comprise a game specific engine module for executing the instructions implementing the specific game on the video gaming machine. The plurality of platform support and utility modules comprise a wagering funds module, a user interaction module, a multimedia output module, and a random number generator module. The plurality of game specific modules and associated data files, the plurality of game specific modules and associated data files comprise a game specific engine module for executing the instructions implementing the specific game on the video gaming machine, a plurality of video image data files combined by the game specific modules to create a sequence of image displayed on the video display; and a plurality of audio sound data files played by the game specific modules to create a series of sounds generated on the audio output module in conjunction with the sequence of image displayed on the video display.



ABSTRACT

An apparatus for compartmentalizing various game instruction modules within an overall architecture of a video gaming machine. The video gaming machine comprises a video display, an audio output module, and a plurality of user input buttons to provide re-usable instruction modules common to all games implemented on a video gaming platform. The apparatus comprises an open-source operating system, a plurality of platform support and utility modules common to all games run on the video gaming machine, and a plurality of game specific modules and associated data files, the plurality of game specific modules and associated data files comprise a game specific engine module for executing the instructions implementing the specific game on the video gaming machine. The plurality of platform support and utility modules comprise a wagering funds module, a user interaction module, a multimedia output module, and a random number generator module. The plurality of game specific modules and associated data files, the plurality of game specific modules and associated data files comprise a game specific engine module for executing the instructions implementing the specific game on the video gaming machine, a plurality of video image data files combined by the game specific modules to create a sequence of image displayed on the video display; and a plurality of audio sound data files played by the game specific modules to create a series of sounds generated on the audio output module in conjunction with the sequence of image displayed on the video display.

# METHOD AND APPARATUS FOR PROVIDING A COMPARTMENTALIZED GAME INSTRUCTION ARCHITECTURE WITHIN A GAMING MACHINE

## FIELD OF THE INVENTION

5           This invention relates in general to a method and apparatus for organizing the game instruction architecture in a video gaming device. More particularly, this invention relates to a method and apparatus for compartmentalizing various game instruction modules within an overall architecture of a video gaming machine to provide re-usable software modules common to all games implemented on a platform which are separable  
10       from game-specific modules common to a specific video game.

## BACKGROUND OF THE INVENTION

          Computer-based, video gaming machines are becoming increasingly commonplace to construct gaming devices such as slot machines, video poker games, and video roulette wheels. These automated video games utilize computing systems  
15       containing software modules to implement gaming logic. These computing systems also utilize computer video display devices to present gaming players with images of the various gaming apparatus being implemented.

          These computer-based gaming systems replace mechanical systems such as slot machines comprising a plurality of rotating wheels and associated mechanical logic. The  
20       computing systems utilize a random number generator to determine a game outcome that statistically appears to be random in nature. The random numbers obtained from a random number-generating module are used to determine which symbols on the wheels of a slot machine are to be displayed when the game concludes a play. Similarly, these random numbers are used to shuffle standard decks of playing cards used in other games  
25       of chance.

          These computer-based gaming machines also comprise software modules which when working together implement the rules of a particular game of chance. For a slot machine, these rules include the pay-out tables used to determine any winnings paid to a player for a particular combination of symbols shown on the rotating game wheels.  
30       Finally, the computer gaming machines comprise software modules which when working

together display a series of images on the display device to simulate the appearance and operation of a gaming machine. These display modules typically comprise both video and audio output modules to provide a game player with a sensory experience comparable to the mechanical gaming machines.

5           Gaming machines that accept wagers and provide winning payouts are under a wide variety of regulatory oversight procedures and provisions from authorities of the various jurisdictions that permit the use of these devices. These oversight procedures and provisions are concerned in part with providing a level of assurance that the games operate in the manner advertised. The behavior of the random number generator, its  
10          relationship to the outcome of the game implemented, and the winning pay-out tables are part of the functions of these gaming devices which are inspected. The procedures for obtaining regulatory approval for each gaming device may be a long, complicated, and expensive undertaking on the part of the gaming machine manufacturer and its customers, gaming establishment operators.

15           At the same time, the gaming establishment operators desire a wide variety of different games to entice game players to play the gaming machines within their establishments. Part of this desire for a variety of different games includes a desire to periodically change the mix of different games present within the establishment. Establishment operators desire to increase the number of gaming devices of a particular  
20          type which induce players to wager larger amounts of funds and/or induce players to play these games for a longer period of time. At the same time, the operators desire to remove gaming devices which to not induce players to wager at profitable rates.

          Computer-based gaming machines appear to address the establishment operators' desires as computing systems are capable of generating a nearly infinite number of  
25          display images and audio combinations. As such, the computer-based gaming machines are capable of being reconfigured from a first gaming device to a second gaming device while utilizing many of the same components in a single configuration. One issue which may prevent the simple reconfiguration of a computer-based gaming machine from one game implementation to a second gaming implementation is the regulatory oversight  
30          approval for each gaming implementation.

The process of obtaining approval for a complete software implementation of a gaming device that operates on a common computer-based gaming platform having a standard set of hardware components may be simplified with a software architecture that structures its software modules into an organization which re-uses a large amount of its software modules to implement functions common to all gaming devices. With such a software architecture, the common software modules would seek approval the first time the modules are used in a gaming machine. For subsequent game machine implementation, the prior approval for the common module would be provided along with any changes implemented to the modules. Additionally, new modules used to implement game specific functions would be identified. As such, the previously approved common modules that may not change in a significant way may obtain approval more quickly and less expensively. The process of obtaining the approval for new gaming implementations may likely focus on the new game-specific modules. Since these modules represent a subset of an entire system, the approval may be obtained more quickly and less expensively.

With the ability to more easily obtain approval for new gaming implementations, gaming establishments are provided with an improved ability to more readily change the gaming devices present in their establishments to satisfy the interests of their gaming players. The present invention provides a software architecture for implementing computer-based gaming machines to address the above problems in prior systems.

### SUMMARY OF THE INVENTION

To overcome the limitations in the prior art described above, and to overcome other limitations that will become apparent upon reading and understanding the present specification, the present invention discloses an apparatus for compartmentalizing various game instruction modules within an overall architecture of a video gaming machine.

The video gaming machine comprises a video display, an audio output module, and a plurality of user input buttons to provide re-usable instruction modules common to all games implemented on a video gaming platform. The apparatus comprises an open-source operating system, a plurality of platform support and utility modules common to all games run on the video gaming machine, and a plurality of game specific modules and associated data files, the plurality of game specific modules and associated data files

comprise a game specific engine module for executing the instructions implementing the specific game on the video gaming machine.

The plurality of platform support and utility modules comprise a wagering funds module for accepting and paying out funds associated with wagers played on the outcome of the games run on the video gaming machine, a user interaction module for generating output display information and accepting user input instructions, a multimedia output module for generating output signals on the video display and audio output module, and a random number generator module for creating the random numbers used to generate the outcome of the games run on the video gaming machine.

The plurality of game specific modules and associated data files, the plurality of game specific modules and associated data files comprise a game specific engine module for executing the instructions implementing the specific game on the video gaming machine, a plurality of video image data files combined by the game specific modules to create a sequence of image displayed on the video display; and a plurality of audio sound data files played by the game specific modules to create a series of sounds generated on the audio output module in conjunction with the sequence of image displayed on the video display. These and various other advantages and features of novelty that characterize the invention are pointed out with particularity in the claims annexed hereto and form a part hereof. However, for a better understanding of the invention, its advantages, and the objects obtained by its use, reference should be made to the drawings which form a further part hereof, and to accompanying descriptive matter, in which there are illustrated and described specific examples of an apparatus in accordance with the invention.

#### BRIEF DESCRIPTION OF THE DRAWINGS

In the following description of the exemplary embodiment, reference is made to the accompanying drawings that form a part hereof, and in which is shown by way of illustration the specific embodiment in which the invention may be practiced. It is to be understood that other embodiments may be utilized as structural changes may be made without departing from the scope of the present invention.

Fig. 1 illustrates a video-based gaming machine according to one embodiment of the present invention;

Fig. 2 illustrates a logical block diagram for a computer-based gaming system used to implement gaming device according to a second embodiment of the present invention;

Fig. 3 illustrates an expanded logical block diagram of the processing system 202 used to implement the gaming device according to an embodiment of the present invention;

Fig. 4 illustrates a system module block diagram of the software utilized to construct the gaming machine according to one embodiment of the present invention;

Fig. 5 illustrates a detailed system module block diagram of the software utilized to construct the gaming machine according to another embodiment of the present invention;

Fig 6a illustrates the relationships between the Java classes associated with a Game Application Context;

Fig 6b illustrates the relationships between the Java classes associated with a Game Applet Context;

Fig 6c illustrates the relationships between the Java classes associated with a Station Context;

Fig 6d illustrates the relationships between the Java classes associated with a Device Layer Context;

Fig 7 illustrates the relationships between the Java classes associated with Multi-Player Classes;

Fig 8 illustrates the relationships between the Java classes associated with Message Classes;

Fig 9 illustrates the relationships between the Java classes associated with Local Message Passing; and

Fig 10 illustrates the relationships between the Java classes associated with Network Message Passing.

## DETAILED DESCRIPTION OF THE INVENTION

In the following description of the exemplary embodiment, reference is made to the accompanying drawings, which form a part hereof, and in which is shown by way of illustration the specific embodiment in which the invention may be practiced. It is to be understood that other embodiments may be utilized as structural changes may be made without departing from the scope of the present invention.

In general terms, the present invention relates to a method and apparatus for compartmentalizing various software modules within an overall architecture of a video gaming machine to provide re-usable software modules common to all games implemented on a platform which are separable from game-specific modules common to a specific video game. The embodiments of the invention described herein are implemented as logical operations in a mass storage subsystem attached to a host computer system having connections to a distributed network such as the Internet. The logical operations are implemented (1) as a sequence of computer implemented steps running on a computer system and (2) as interconnected machine modules running within the computing system. This implementation is a matter of choice dependent on the performance requirements of the computing system implementing the invention. Accordingly, the logical operations making up the embodiments of the invention described herein are referred to as operations, steps, or modules. It will be recognized by one of ordinary skill in the art that these operations, steps, and modules may be implemented in software, in firmware, in special purpose digital logic, and any combination thereof without deviating from the spirit and scope of the present invention as recited within the claims attached hereto.

Fig. 1 illustrates a video-based slot machine according to one embodiment of the present invention. A gaming device 100 is used to implement the video-based slot machine and comprises a video display device 101, game-related artwork 102, user controls 103, a hopper 104, and wager input modules 105. Within the gaming device 100, a computing system is utilized to operate the gaming device 100 and generate the video images displayed upon the video display device 101.

Now referring to Fig. 2, a logical block diagram for a computer-based gaming system used to implement gaming device 100 is illustrated. The computer-based gaming

system 200 comprises a plurality of mass storage devices 211-212, a video connection 201 to video display device 102, and a plurality of user interface units 220. The user interface units 220 comprise a hopper 221, a coin acceptor 222, a set of meters 223, a button panel 224, and a bill validator 225.

5           The hopper 221 provides a game player with tokens or coins representing something of value returned when the game player places a wager on an outcome of a play of the game resulting in a successful outcome for the game player. The hopper 221 may be configured to return these winnings from the outcome of each play of the game. The hopper 221 may also be configured to return a current balance of funds maintained  
10       within the gaming device 100 upon receiving a command from the game player. In the latter embodiment, the winnings from any given outcome from a play of the game are added to a current balance maintained within the processing system 202. This current balance may be displayed upon one of the set of meters 223. The game player may also place wagers on the outcome of the play of the game from the current balance.

15           The coin acceptor 222 and the bill validator 225 provide the game player with ability to input coins, tokens, or paper bills representing currency into the gaming device 100 for use in making wagers on the outcome of a play of the game. The game player may input coins or tokens into the coin acceptor 222, and may input bills into the bill validator 225 before each play of the game is initiated. The player may also input these  
20       coins or other forms of money into the gaming device 100 to create the current balance, which is used to wager on a series of outcomes from the game.

          The button panel 224 comprises a plurality of user input buttons used to control the operation of the game. These buttons may comprise a button to start a game, a button to return the current balance to the hopper 221, and one or more buttons to determine the  
25       amount of funds to be wagered on the outcome of the next play of the game. The game player depresses these buttons to provide a command to the processing system 202 how to proceed with the operation of the gaming device 100.

          Fig. 3 illustrates an expanded logical block diagram of the processing system 202 used to implement the gaming device 100. The processing system 202 comprises a main  
30       processing module 301, a video control module 302, a storage control module 303, and an I/O control module 305. Each of these modules are coupled together using a system

communications bus 304. The video display device 102 is coupled to the video control module 302 to display images generated by the processing system 202 to the game player. The mass storage devices 211-212 are coupled to the storage control module 303 to provide the processing system with mass memory storage for the data and program files needed to implement the gaming device 101.

According to the preferred embodiment, the processing system 202 is implemented using a computing system typically referred to as a personal computer. The computing system typically uses a PCI and ASI or PC-104 type system bosses to implement the system communications bus 304. This choice to implement the preferred embodiment permits the main processing module 301, the video control module 302, and the storage control module 303 to be implemented with a wide variety of commonly available system components. This choice also permits the periodic improvement of the processing system 202 with the upgrade of one of these modules as new and faster computing modules become available. The preferred embodiment utilizes a main processing module 301 based upon a Pentium II processor manufactured by the Intel Corp. One of ordinary skill in the art will recognize that this processing unit may be based upon any number of alternate processing units manufactured by Advanced Micro Devices and other manufacturers as well as a PowerPC processor manufactured by IBM Corporation and Motorola.

Additionally, the preferred embodiment of the processing system 202 utilizes a commercially available operating system, such as the Linux operating system available from Walnut Creek CD-ROM, as the 'Slackware' distribution. The operating system is stored upon mass storage disk 212 for execution with the main processing module to provide device driver support for the peripheral devices used to implement the system.

The I/O control module 305 is a custom logic interface module for providing connections between the user interface units 220 and the processing system 202. Each of the various user interface units 220 comprise one or more electro-mechanical device that interacts with the game player. These units 220 accept electrical signals generated by the I/O control module 305 instructing the units to perform an operation. These units 220 also generate electrical signals accepted by the I/O control module 305 for use by the software modules executing within the main processing module 301.

Now referring to Fig. 4, a system module block diagram of the software utilized to construct the gaming machine is illustrated. The software 401 of the gaming device comprises a plurality of software modules including operating system modules 402, platform support and utility modules 411, and game specific modules 412. As stated  
 5 above, the operating system modules 402 comprises a version of the Linux open source operating system. One skilled in the art will recognize that these modules may alternatively be implemented using other operating systems such as Windows 98 and NT from the Microsoft Corporation, other versions of Unix, and the MacOS operating system from the Apple Corporation without deviating from the spirit and scope of the present  
 10 invention.

The platform support and utility modules 411 comprise a set of modules to provide the gaming device 100 with interaction with the user interface modules 220 as well as providing any audio and video output support not provided within the operating system modules 402. These modules are common to all of the different games that are  
 15 implemented upon the gaming device 100.

The game specific modules 412 comprise a set of modules needed to implement a particular version of a video-based gaming machine. The modules will include gaming instruction modules, audio files, and video files utilized to present the game-related and other images and audio information to the game player upon the display device 102.  
 20 These modules may also comprise modules to present advertising and announcement data in both audio and video formats to game players when a particular play of the game is not being conducted.

Fig. 5 illustrates a detailed system module block diagram of the software utilized to construct the gaming machine. Once again, the software modules 400 of the gaming  
 25 device 100 comprise operating system modules 402, platform support and utility modules 411, and game specific modules 412. The platform support and utility modules 411 comprise a hopper module 511, a bill and coin acceptance module 512, a meter display and register module 513, a buttons module 514, an audio output module 515, a video output module 516, and random number generator 517. The game specific modules  
 30 412 comprise a game specific engine module 501, video image data files 502, and audio sound data files 503.

The hopper module 511 controls the operation of the hopper user interface unit 221 to return winnings to game players from wagers made on various plays of the game. The hopper 221 returns coins, tokens, and other forms of currency upon receiving appropriate commands from the hopper module 511. The hopper module 511 interacts  
 5 with other modules to determine the proper amount of winnings to be provided by the hopper 221 and instructs the hopper 221 to return the proper number of coins in appropriate denominations to provide the winnings.

The bill and coin acceptance module 512 interacts with the coin acceptor 222 and bill validator 225 user interface units to accept currency for use in making wagers on the  
 10 outcome of a particular play of the game. The bill and coin acceptance module 512 determines the proper numerical value for the bills and coins inserted into the user interface units. The module 512 will interact with the meter display and register module 513 to add the value of each coin and bill inserted into an accumulator register that holds the amount of funds available for use by the game player for making wagers on the  
 15 outcome of particular plays of the game.

The meter display and register module 513 maintains the accumulator register that holds the amount of funds available for use by the game player for making wagers on the outcome of particular plays of the game. This module 513 generates the signals needed to display this value upon one or more meters 223 and displays 102 on the gaming device  
 20 100 for viewing by the game player. The module 513 will also maintain and display other values such as the number of plays completed by the game player, an amount of a jackpot award available to be won, and the amount of winnings awarded by the gaming device over a specified period of time.

The buttons module 514 interacts with the buttons user interface unit 224 to accept  
 25 inputs to the gaming device 100 from the game player. The module 514 detects when one of the buttons are depressed. The module 514 decodes the button and generates the corresponding command to an appropriate module responsible for implementing the desired instruction. The buttons module 224 is also responsible for generating any signals necessary to illuminate and flash lights that may be part of the buttons module. In one  
 30 possible embodiment, lights associated with the buttons which are encoded to generate valid commands to the game at a given time may be illuminated. The other buttons

would not be illuminated to provide an indication to the game player which buttons may be depressed to initiate action. The se lights associated with the various buttons on the buttons panel 224 may be flashed when the game in not being played as to call attention to game to game players present in the gaming establishment.

5           The audio output module 515 and the video output module 516 generate the appropriate output signals to generate signals necessary to display a sequence of images upon video display unit 102 along with corresponding audio signals to provide the game player with a desired experience while playing the gaming device 100. These data output modules perform general output functions necessary to output game images and audio, 10 advertising images and audio, and general announcement images and audio. Other modules within the software system 401 generate and format the audio and video data into the proper format necessary for output by these routines.

          The random number generator 517 generates a pseudo-random number for use by the specific game engine module 501 to generate the outcome of the gaming device. The 15 random number generator module 517 is based on a method described by Knuth in volume 2 of the Art of Computer Programming. In this method, random numbers are drawn from an initial random number generator module 517 implementation to populate a table of sufficient size (1000 by default) and these numbers are used as indices back into the table to draw and replace the numbers in the table. This can be shown to remove 20 banding.

          In addition, a background process draws numbers from the table at frequent but imprecise 100 ms intervals and discards them, making them unavailable to the game. The startup second of random number generator module 517 class loading is used as the seed. These 2 elements render number selection by the game truly random. The random 25 number generator module 517 is invoked at least once during each polling cycle to insure compliance with the Nevada regulation requiring at least 100 background draws per second.

          In the preferred embodiment, two implementations of this method are included: a pure Java implementation where the random number generator used as the basis is the one 30 included with Java and a native implementation employing a Linear Shift Register to generate the underlying random sequence. This implementation saves it's seed

periodically in non-volatile RAM and restores it at startup. This random number generator module 517 is currently implemented in a preferred embodiment as a Linear Shift Register Random Number Generation, with coefficients  $a = 16807$ ,  $c = 0$ ,  $m = 0x7ffffff$ ,

$$\text{where } R(n+1) = ((a * R(n)) + c) \bmod m.$$

5 Further information on Linear Shift Registers is available in Applied Cryptography, Second Edition by Bruce Schneier; John Wiley and Sons: 1996.

The game specific module 412 comprises video image data modules 502 and audio sound data modules 503 for output by the above two output modules during the playing of the game. These data modules comprise audio and video data files which may  
10 be simply played for viewing and listening by the game player. These data modules further comprise audio and video files which may be combined together to synthesize the output audio data streams.

For example, the video data modules may comprise a series of images that represent the reels of a slot machine in all of the possible positions. The video data files  
15 may further comprise a set of background images and a set of slot machine images that are combined with the images of the reels to create a composite image that is presented to the game player. At the same time, audio files relating to the rotation of the wheels may be played while the images illustrate rotating wheels. Similarly, audio files for celebrations and disappointment may be played at the conclusion of the play once the  
20 reels stop rotating to provide the game player with an enhanced game experience.

Other examples of this process comprise a set of images of playing cards, a set of card table arrangements, a set of dealer's hand images in various positions, a set of audio files of shuffling cards, a set of audio files of cards being dealt, and audio files for celebrations and disappointment being combined to create a video-based card gaming  
25 machine. One skilled in the art would recognize how other games of chance such as craps, roulette, and like games may be constructed from the synthesis of many smaller images and audio files.

The game specific engine module 501 comprises a set of modules to implement the specific game desired to be run on the gaming device 100. These modules would  
30 define the game, the relationship between the elements such as cards and reels, and the

inputs from the game player to initiate and complete a play of the game. These modules are the only instruction modules needed to be changed when the gaming device is reconfigured from one game to another.

5 In the preferred embodiment, the game specific modules 412 and the platform support and utility modules 411 are implemented in an object oriented language such as Java from Sun Microsystems.

Software consists of a bootstrap ROM-DOS system and Linux kernel in EPROMs or boot record of a CD-ROM 211, with Linux system libraries, Java interpreter, game code and sound and image files on the CD-ROM 211. ROM-DOS is booted initially;  
10 ROM-DOS loads Linux along with a stripped-down root file system, using a GNU Public License utility LOADLIN.EXE. Linux mounts the CD-ROM either directly, or via NFS (Network File System) over Ethernet in the case of multi-player games. System libraries, X (if the system supports graphics) and the Java interpreter are loaded from the CD-ROM 211. The Java interpreter loads and initializes the game code and any additional libraries  
15 that may be required, including the Random Number Generator module 517. The game code loads meter values, events and play history from NVRAM and initializes an Online Accounting System, if appropriate.

The game then loops until it is shut down, either by a power down, or some non-recoverable error. While looping, the game monitors the required security switches,  
20 credit input devices and game-control devices.

When a switch is open or there is a severe error on any device, the condition is displayed and may only be cleared by opening at least one door and subsequently closing all open doors. Jackpot alarms or hand-pay requests may be cleared with the attendant key. When an attendant key is in the "on" position, audit mode is entered, where meters,  
25 events, the bill-log and play history may be viewed.

In the case of a required hand-pay, an alarm is set that may only be cleared by pushing a cash-out button, with the attendant key in the "on" position. A hand-pay is required whenever some statutory cash amount must be paid or a hopper pay-out limit is exceeded.

Included in the root file system is a utility, whichplayer. If the station is a remote player (as in multi-player blackjack), the user is given an opportunity to use this utility to set the player ID to a number from 1 through 5, using whichplayer. Once the player ID has been set, whichplayeronly runs if the attendant key has been turned to its "on" position.

Within whichplayer, the clear switch must be set to its "on" position inorder to set the player ID. The player ID is then set by means of the trouble switch; each time the trouble switch is pressed, the player IDtoggles to the next value. The utility exits when the clear switch is set to its "off" position, or the attendant key is turned to the "off" position and a player ID has been selected.

Next, an attempt is made to initialize the network connections. In the case of a remote player, this can only be done after successfully setting the player ID. If more than one player has the same ID, one or both will hang at this point; neither will have a reliable network connection. In the case of the dealer or a single-player game, network initialization failure is not fatal, but if the dealer fails to initialize, no communication takes place with the remote players and they will ultimately fail to mount the CD-ROM, and exit. In the case of a remote player, excution will halt immediately if a networkconnection cannot be initialized.

In the boot image and game-specific executables on the CD-ROM are verified at this time. When the CD-ROM is created, a file named "medium.verifcation" is placed on the CD-ROM. This file contains the 16-bit CCITT CRC values and sizes of the files. Before the game can be started, this file is re-created in memory and compared to the CD-ROM. If any discrepancy is found, the game will not be loaded.

The current implementation of the software requires the use of X as the presentation layer for graphical games. This is required by Java AWT (Abstract Windowing Toolkit). If X fails to load correctly, execution halts with appropriate error messages on the video display. Once X has loaded, error messages may be hidden by the X display. To view them, a standard PC keyboard may be attached to the motherboard. By typing CTRL-ALT-F1, the text console appears. The user may then type SHIFT-PageUp and SHIFT-PageDown to view the text. Immediately after X is loaded, the JVM

(Java Virtual Machine) starts and begins to load either a graphical shell for the game code (if there is a video display) or a non-graphical application for controlling the game loop.

The game-specific code is derived either from a non-graphic class, powerpit.mains.BaseGame or a graphical class derived from powerpit.mains.GameApplet (a subclass of java.applet.Applet). The class BaseGame implements basic station loading, message and event handling, common to all games. It also processes some parameters such as hopper limits (HAND-PAY), game denomination (DENOMINATION) or maximum bet allowed (MAX\_BET). The class GameApplet uses an instance of BaseGame to provide this functionality for graphical game stations.

In addition, GameApplet adds generalized graphical functionality for implementation of displays for game-play, audit screens and alarm conditions. In any form, the game portion of the software reads the required parameters, attempts to load a station and log files (if defined), then loops until stopped by some error condition, responding to inputs from the player devices or network messages (if multi-player).

If a graphical display is used, as on a dealer station for multi-playergames, the game applet must be launched indirectly through the graphics shell (similar to a WWW browser), implemented by powerpit.mains.ShellFrame and powerpit.mains.MainFrame. An otherwise empty main() routine in ShellFrame is used to read the command-line arguments and pass them to a instance of MainFrame, which implements the required context for a game applet.

Configuration is by means of a configuration file in the following format:

1. Comments are either totally blank or begin with "/" (2 slashes as with a Java or C++ comment);
2. Keywords (parameter names) begin in column 1;
3. Parameter values are seperated from a keyword by at least 1 blank or tab; the values continue until a new keyword (non-comment) or end-of-file;

4. Parameter values may contain non-nested, quoted strings. E.g: PARM1 a "b c" d is valid, but PARM2 a "b c "d e"" is not; and

5. If parameter names are duplicated, only the first set of values is used.

The name of the configuration file is passed on the command line as either a  
 5 relative or absolute path. The configuration file resides, with the game code on the CD-ROM locked in the logic cage. Note that many of the parameters (e.g: GAME\_APPLET, STATION, HOPPER) take fully qualified Java class names as their arguments. This simplifies manufacturing, testing and regulatory submission by indicating the actual code specific to each configuration. Fig. 6a-6d contain Unified Modeling Language (UML)  
 10 diagrams of the relationships between the main Java classes used in all games.

Fig 6a illustrates the relationships between the Java classes associated with a Game Application Context. The Game Application Context comprises a Non-Graphical Game class 620 which interacts with liob Device 625, JaudioCip 623, OtherUtility class 622, ConfigurationFile 621, and BaseGame 610. BaseGame 610 interacts with classes  
 15 GameState 615, JLog 614, Station 613, Queue 612, JRandom 611, and BaseGameInterface 601. BaseGameInterface 601 interacts with GameInterface 602, which in turn interacts with MessageInterface 603 and ParameterInterface 605. MessageInterface 603 interacts with Message 604. JAudioClip 623 interacts with AudioClip 624 from java.applet.

20 Interface powerpit.mains.BaseGameInterface 601 is an interface required for all game classes. The class includes:

attendantKey() Return attendant key value (on/off)  
 destroy() Halt game execution  
 25 getDenomination() Get game denomination  
 getMaxBet() Return maximum bet allowed  
 getMinBet() Return minimum bet allowed  
 getName() Get unique class name of game  
 getNumberOfPlayers() Return number of players (required for GameData)  
 getRNG() Return reference to random number generator for game (may be null).  
 30 getSequence() Get game sequence identifier  
 getState() Get game state from GameData or subclass  
 lastAlarm() Return most recent alarm condition

Interface powerpit.mains.GameInterface 602 is a public interface. GameInterface  
 35 extends MessageInterface and ParameterInterface.

Interface powerpit.mains.MessageInterface 603 is a public interface.

MessageInterface includes:

5           processMessage(Message) process a message sent to this class

Class powerpit.messages.Message 604 implements abstract base class for system messages and provides the standard methods to allow queuing, logging, display and network transmission of all internal data representing events within the game system along with the resulting overall game state. The class includes:

copyOf(Message) make a copy of an arbitrary message  
 date(long) return formatted date string  
 display() display message data in ASCII  
 expand(byte[]) expand packed message data into actual message  
 15    getClassName(byte[]) Find class name in packed data, so we can instantiate the  
       message  
 loadMessage() test routine for incorporation in derived class tests  
 pack() return packed data for network transmission  
 testMessage(byte[]) test routine for incorporation in derived class tests  
 20    unpack(byte[]) decode packed data after network transmission

Interface powerpit.mains.ParameterInterface 605 is a public interface for ParameterInterface. It included getParameter(String) return ASCII string value of specified parameter.

25           Class powerpit.mains.BaseGame 610 is a non-graphical game base class. It is used as an adapter by GameApplet; extended by any non-graphical game; e.g. spinning reel slot driven by stepper motors. The class includes:

adjustStatus() set game state to idle if there are no players available, ready to play  
               the game.  
 30    alarmMessage(AlarmMessage) handle various game tilts  
 attendantKey() return attendant key value  
 audit() toggle audit flag  
 auditCheck() audit check performed in player game loop before each game cycle  
 autoplay() return true if game is in autoplay mode  
 35    canPlay() Return an array of references to those Player Stations with credit  
               available for play.  
 clearSwitch() Return clear switch value  
 debugPrint(String)  
 debugPrintln(String)  
 40    destroy() end game and clean up any loose threads

emptyQueue() Reset game event queue and force garbage collection  
 emptyQueue(int) re-call emptyQueue repeatedly for ms milliseconds  
 getAlarmList() return ASCII descriptions of alarms since the game was last  
     playable  
 5     getDealer() Get the dealer station  
       getDenomination() Get the game denomination  
       getLog() Get the state log  
       getMaxBet() Get the maximum bet  
       getMinBet() Get the minimum bet  
 10     getName() Get unique class name of game  
       getNumberOfPlayers() Return number of players (required for GameData)  
       getParameter(String) return configured parameter as an ASCII string  
       getPlayer() Get the player station  
       getPlayers() Get the player station stubs for a multiplayer game  
 15     getRNG()  
       Return reference to random number generator for game (may be null).  
       getSequence()  
       Get game sequence identifier  
       getState()  
 20     Get game state from GameState or subclass  
       getStateData() Get the whole state data block  
       getStation() Get the station (may be player or dealer)  
       init() Do minimum game initialization: get STATION, DENOMINATION  
             MIN\_BET, MAX\_BET, HAND\_PAY, GENERATOR and STATE\_LOG  
 25     parameters.  
       isAlarmed() test for current tilt of any kind, other than jackpot  
       jackpot() test for current jackpot tilt  
       killGenerator() Terminate with extreme prejudice; should be used only in games  
                     and by Stopper class.  
 30     lastAlarm() return most recent alarm condition  
       nextMessage() This is the individual Game's only window into the message queue.  
       poll() perform periodic polling of Station/device interfaces  
       processMessage(Message)  
       resetTournamentCredits(long)  
 35     resetTournamentTimeout(long)  
       setStateData(GameData)  
       Replace state data  
       setTimeout(int)  
       implement timeout handling in a standard way; this helps insure that timeouts will  
 40     not expire because of asynch garbage collection.  
       start()  
       Recover saved state from log if possible; finish generic setup.  
       startStation()  
       Start the station  
 45     startTournament() called by player station on the first valid bet, to start timer  
       timeToStop() Used by individual Game to read m\_fStopNow flag.  
       tournament() Is the game in tournament mode?  
       tournamentCredits() set tournament credits

tournamentIsOver() tournament play ends when credit goes to 0, or when timer  
 expires  
 tournamentSecondsLeft() how many seconds are left in tournament mode  
 tournamentStarted() has the tournament round started? called by game (or  
 5 anybody else that needs to know)  
 tournamentTimeout() has the tournament round timed out?  
 troubleSwitch() return trouble switch value  
 waitingForHandpay() are we waiting for a handpay?

10 Interface powerpit.chance.JRandom 611 is a public interface. JRandom 611  
 Provide a standard interface for random number generation. Allows distribution of a  
 default generator for development, then use of approved generator on gaming machines.

The class includes:

15 display() Toggle display of random numbers as they are drawn.  
 getModulus() Retrieve modulus for random numbers.  
 nextInt() Draw the next random number.

Public class powerpit.mains.Queue 612 provides storage for game events from the  
 machine, such as button presses or play step information in multi-player games. The class  
 20 includes:

25 get() Retrieve message from queue.  
 getLeft() Get space remaining in queue.  
 getSize() Return total capacity of queue.  
 put(Object) Put message in queue  
 reset() Empty queue

Class powerpit.devices.Station 613 is a public abstract class. Station implements  
 GameInterface and StateInterface Base class for player and dealer stations. The class  
 includes:

30 candleOff() manual control of error candle by game  
 candleOn() manual control of error candle by game  
 disable() disable station, as for alarm; stop generation of game-related events,  
 other than tilts  
 disableHardMeters() temporarily stop rolling hard meters to prevent glitching  
 35 displayLog(int) provide ASCII text of specified log contents  
 enable() enable station to generate all game-related events, such as button-presses  
 enableHardMeters() enable hard meter pulses  
 getParameter(String) as above  
 getState() Return integer representing station state  
 40 isClient() let the game know if it's a serial client for test  
 isConsole() let the game know if it's a serial console for testing  
 offline() Take station offline in multi-player game  
 poll() poll attached peripheral devices and insert events in game queue if required,

by invoking game's processMessage() method.  
 powerDown() process power-down signal from io board  
 print(String) print something if ticket-printer is attached  
 processMessage(Message) see above  
 5 register(IioDevice) used by game to register specialized devices with the iio;  
     can only be invoked while station is offline, before first call to enable ().  
 resetLogs() for polymorphism; should be overloaded by any subclass with logs in  
     addition to meters and events - overloaded by player station for play  
     history and bill logs; invoked by powerpit.mains.Audit to display all logs  
 10 in audit mode.  
 resetMeters() reset any logs that need it  
 shutdown() Shut station down at game exit.

Class powerpit.mains.audit.JLog 614 is a class that includes:  
 15 display() provide stored data in ASCII format for display  
 NMI() set flag to indicate write to logs is forbidden during power-down sequence  
     clean-up  
 NMIReceived() used by any class that needs to know power-down sequence has  
 20 been initiated  
 recover() Recover operation for specific object type to be logged.  
 save(Object) Abstract method must be implemented by each subclass - save  
     message to log.

25 Class powerpit.mains.GameState 615 is a base class containing integer values,  
 indicating game state. Used by GameData subclasses to indicate last saved state for  
 recovery after power-down or other catastrophic failure.

Class powerpit.mains.ConfigurationFile 621 read values from configuration file  
 30 on CD-ROM and store them so getParameter() calls from other game classes can be  
 serviced.  
 checkVersion() check current configuration against value stored in non-volatile RAM  
 getParameter(String). The class includes the following:

35 isNewVersion() return true if version has changed; called by AuditCheck() in  
     main game loop  
 resetVersion() reset version information

Other Utility classes 622 are game specific utilities required for individual  
 40 implementation of game "personality". E.g. card decks, roulette wheels, combination  
 tables for slots.

Class powerpit.mains.JAudioClip 623 implements abstract base class java.applet.AudioClip from Sun Microsystems Java 1.1 specification. The class adds capability to play Microsoft WAV files. The class is used by game software, in conjunction with separate C-language sound server to play sound files.

5       Public Class java.applet.AudioClip 624 is a Sun Microsystems abstraction of a sound file.

Interface powerpit.devices.IiobDevice 625 is an abstract base class for devices that need to register with the IIOB. First it is created in an OFFLINE state by a station. Next, the station causes the device to register with the IIOB. The IIOB enables the device when  
10 it is ready to accept data. The IIOB then disables the device when a power-down notification is received or the physical device is otherwise disabled. Finally, getData() is invoked whenever there is data ready for the registered object. The class contains the following applets:

15       disable() Invoked by IIOB to stop device.  
      enable() Invoked by IIOB to bring device online.  
      getData() Invoked by IIOB when data has been received on a line registered to the object.  
      register(IIOBBase) Invoked by Station or other device "owner" to cause device to register with IIOB game software for configuration on the physical iioB.

20       Fig 6b illustrates the relationships between the Java classes associated with a Game Applet Context. The Game Applet Context comprises a GameApplet class 640 that interacts with classes Applet 642 from java.applet, GraphicalGame 645, BaseGame 641, MainFrame 630, and BaseGameInterface 635. MainFrame 630 interacts with classes  
25       AppletContext 631 from java,applet, AppletStub 632 from java,applet, Window 633 from java,applet, and ParameterInterface 634. GraphicalGame 645 interacts with classes liobDevice 651, JAudioClip 652 from java.applet, and Other Utility Classes 653. JAudioClip 652 ineracts with class AudioClip 654 from java.applet. BaseGameInterface interacts with class GameInterface 644, which in turn interacts with ParameterInterface  
30       646 and MessageInterface 647. MessageInterface 647 itself interacts with class 648.

Public Class powerpit.mains.MainFrame 630 extends Window implements AppletContext, AppletStub, ParameterInterface. Usage is: MainFrame.main (<configuration>), where configuration: configuration file name. An object of class MainFrame is created by JShellFrame, which retrieves the configuration file name from

its command line. Information to launch the Game applet comes from the configuration file. Note that the members other than main() and the appletContext and appletStub methods are private; the class is final and can have no subclasses. It implements the graphical environment normally supplied by a "browser" so a GameApplet can execute.

- 5           Public Interface java.applet.AppletContext 631 is a Sun Microsystems abstraction of the basic context required for graphical applet execution.

Public Interface java.applet.AppletStub 632 is a Sun Microsystems abstraction of stub functions required for graphical applet execution.

- 10           Public Class java.awt.Window 633 is a Sun Microsystems abstraction of an execution window in any GUI.

Public abstract class GameApplet extends Applet 640 implements Runnable, GameInterface and BaseGameInterface. This is the abstract base class for all games with graphics. Note that specific game subclasses of this class are configured by the main (or browser) from the configuration file. The class includes:

- 15           adjustStatus() This routine checks whether any players are available for a game and sets the game state to idle if there are not.  
             alarm() display alert with last alarm  
             alert(String) Display alert text over existing background.  
             attendantKey() return attendant key value  
 20           audit() instantiate powerpit.mains.audit.Audit () to handle player audit screens  
             auditCheck() audit check for meter consistency performed in player game loop before each game cycle  
             canPlay() Return an array of references to those Player Stations with credit available for play.  
 25           clear() Redraw default background, foreground, cursor, alert.  
             clearAlert() Clear alert message for specific screen segment.  
             clearSwitch() return clear switch value  
             clipImage(Image, int, int) Clip image frames for complete tiling, as for animation, card decks, etc.  
 30           clipImage(Image, int, int, int, int) Clip image to specified rectangle.  
             cursor(Image, int) Add cursor image. E.g. hand to place bets or rake in dice or roulette game.  
             destroy() exit game  
             draw(Image, Point) Add an image, without clearing the screen at the specified  
 35           location.  
             drawAlert(Graphics) draw alert in lower right corner for "door-closed" type messages; otherwise, draw normal alert box.  
             emptyQueue() empty the game event queue

emptyQueue(int) empty the queue repeatedly for ms milliseconds  
 foreground(Image, Point) Add foreground image with upper left corner at the  
 specified point; foreground will not be revealed until after the next time  
 clear() is invoked.

5      getDenomination() Return game denomination  
        getMaxBet() return maximum bet allowed; note that this should never be called  
        unless at least one player was previously enabled.  
        getMinBet() return minimum bet allowed; note that this should never be called  
        unless the player was previously enabled.

10     getNumberOfPlayers() Return number of players (required for GameData)  
        getRNG() Return reference to random number generator for game (may be null).  
        getSequence() Return game sequence (from m\_state game data) for player win  
        data.

       getState() Return integer game state value.

15     hideCursor(int) Stop displaying the cursor.  
        increment() Increment game sequence.  
        init() Recover previous state information; connect to dealer and player stations and  
        required devices.

       isAlarmed() return true if game has unresolved tilts.

20     isAlive() Is the game able to function normally? Used by MainFrame to determine  
        whether to exit completely or not.

       isJackpot() has the current game ended in a jackpot award?  
        jackpot() if the current game ends with a jackpot award, handle the clear procedure

       lastAlarm() return most recent alarm condition

25     loadImage(String) Encapsulate robust, synchronous image loading.  
        moveCursor(Point, int) Move cursor to a new position.  
        nextMessage() This is the individual Game's only window into the message queue.

       paint(Graphics) overload paint() method from

       poll() cause local and network inputs and outputs to be serviced

30     processMessage(Message) The method that gets called to distribute button and  
        play messages into the messageQueue so the game loop can examine them;  
        processes commands and alarms locally; ignores others.

       refresh() redraw screen with current background, foreground, alert and cursor.

       repaint() Update the screen (send it or repaint it, as appropriate).

35     run() Abstract method; implements game event loop; i.e. poll() and game-specific  
        methods are called here; alarm (tilt) methods and auditCheck() method for  
        meter auditing are called in this loop to run game as a finite state  
        machine.

       setStateData(GameData)

40     Replace state data  
        setTimeout(int)  
        implement timeout handling in a standard way  
        showCursor(int)  
        Show cursor (if cursor image and position are set).

45     start() Let the game begin! Invokes the specific game's startGame() method.  
        startGame() Game-specific initialization and recovery routines.  
        stop() Pause the runner thread.  
        timeToStop() Used by individual Game to read m\_fStopNow flag; signals receipt

of shutdown or power-down notice  
 tournament() is the game in tournament mode  
 troubleSwitch() return trouble switch value  
 update(Graphics) overload Applet method to stop flicker

5

Java.applet.Applet 642 is a Sun Microsystems generalized implementation of a limited, secure application framework for execution within a well-defined "browser" environment.

Fig 6c illustrates the relationships between the Java classes associated with a Station Context. The Station Context comprises a Station class 660 that interacts with classes SwitchHarness 661, MeterModel 662, Candelabra 663, DealerStation 670, Player Station 669, JLog 668, Device 680 (which in turn interacts with GameInterface 682), liobDevice 667, IIOBase 666, and StateInterface 665. GameInterface 664 interacts with MessageInterface 686 and ParameterInterface 684. MessageInterface 686 interacts with Message 685. IIOBase 666 interacts with classes liobDevice 667 and message 683. LiobDevice 667 interacts with classes IIOBase 666 and Device 680, which in turn interacts with GameInterface 682. DealerStation 670 interacts with classes PlayerStub 671 (which in turn interacts with PlayerInterface 672), ServerSocket 673, ClientSocket 674, and Connection 678. PlayerStation 669 interacts with classes ButtonPanel 675, OnlineSystem 676, NetworkPlayer 677, and PlayerInterface 679, which in turn interacts with StateInterface. Finally, ServerSocket 673, NetworkPlayerStation 677, and Connection 678 all interact with ClientSocket 674.

Class powerpit.devices.SwitchHarness 661 is a public abstract class. SwitchHarness extends Object Describes the station switch harness for a game. It is subclassed by differing machine types to handle system events associated with the portion of the wiring harness devoted to security switches; e.g.: powerpit.slots.NVSMSwitchHarness is used to specify the video slot switch operations. The class includes:

addSwitch(int, boolean, int, int) add another switch; should be used only in constructor  
 poll() poll the switches when re-enabled after alarm  
 register(IIOBase) register all the switches with iioB

Class powerpit.devices.MeterModel 662 is a public abstract class. MeterModel extends Object implements ParameterInterface, an abstract meter model for a Station. It

Instantiates soft meters for coin-in, coin-out, drop, credit, bet, paid, games played and games won. Hard meters and soft meters both incorporate the functionality required for any meter. It is subclassed by differing machine types to handle meter updates and system events associated with the portion of the wiring harness devoted to meters; e.g.:

5 powerpit.slots.NVSMMeterModel and powerpit.slots.COSMMeterModel are used to specify the video slot meter operations for Nevada and GLI/Colorado jurisdictions, respectively. The class includes:

- addCoinOut(int) add special meter for physical coin-out to interact with hopper; called only by constructor.
- 10 addHard(int, int) Add hard meter to model; called only by constructor
- addSoft(int) Add soft meter to model; called only by constructor
- addToUpdate(Meter) Add meter to updated list for logging (once per polling cycle)
- adjustCredit(CreditMessage) Adjust meters for credit.
- 15 adjustMeter(MeterMessage) Process meter data.
- cancelGame() Cancel current game.
- cashing() tell the station whether we're cashing out or not
- cashout() Update meters for cashout.
- disable() disable hard meters for alarm
- 20 disableHardMeters()disable hard meters in game loop
- displayMeters() provide ASCII text display for meter values
- enable() enable hard meters after alarm
- enableHardMeters() re-enable hard meters in game loop
- endGame(long) Adjust meters for end of game.
- 25 findMeter(int) find an arbitrary meter object, based on type as defined in abstract class Meter; used internally and by Station.
- getBet() Return value of bet meter.
- getCashIn() Returns total value of cash in from all sources.
- getCashOut() Returns total value of cash out from all sources.
- 30 getCoinIn() Return value of (nevada) "coin in" (cumulative bet) meter.
- getCoinOut() Return value of (nevada) "coin out" (cumulative win) meter.
- getCredit() Returns value of credit meter.
- getCreditSince() Returns credits since last game
- getEscrow() Return value of escrow meter.
- 35 getPaid() Returns value of paid meter.
- getParameter(String)
- getUpdated() Used by Station to retrieve updated meters, when sending to dealer in multiplayer game , or by any Station for logging at end of polling cycle.
- handpay() clear handpay-required tilt by transferring credits from credit meter.
- 40 increaseBet(long) Adjust meters for bet.
- increment(int, long) Increment an arbitrary meter by type
- ramClear() clear meters in NVRAM
- recover(MeterMessage[]) Recover the meters from saved data; shows a message on the console if there is a missing meter.

register(IIOBase) Register the hard meters with the iioB  
 reset(int, long) Reset a meter  
 resetMeters() reset all meters to 0 after a ram-clear  
 resetTilts() reset hopper tilts  
 5 startGame() Updates games played.  
 stopRolling() clear m\_fRolling flag for i/o board reset  
 undoBet(long) Cancel bet.

Class powerpit.devices.Candelabra 663 is a public abstract class. Candelabra  
 10 extends Object and describes the station candles for a game. It is subclassed by differing  
 machine types to handle system events associated with the portion of the wiring harness  
 devoted to candles; e.g.: powerpit.slots.NVSMCandles and  
 powerpit.blackjack.NVBJDCandles are used to specify the candle operations for video  
 slot candles and blackjack dealer candles, respectively. The class includes:

15 addCandle(int, boolean, boolean) add another candle; should be used only in  
 constructor  
 attendantOff() dim attendant/service/change candle  
 attendantOn() light attendant/service/change candle  
 deselect() dim ready-to-play candle in multi-player games  
 20 errorOff() dim error candle  
 errorOn() light error candle  
 register(IIOBase) register all the candles with iioB  
 select() light ready-to-play candle in multi-player games  
 winOff() dim win candle  
 25 winOn() light win candle

Interface powerpit.devices.StateInterface 665 provides a state interface required  
 for DealerStation, PlayerStation, PlayerStub, NetworkPlayerStation - defines consistent  
 integer values for state interfaces in various station types. The interface includes:

30 disable() Disable station, as for tilt.  
 enable() Enable station; called by game after tilt is cleared.  
 getState() Return station state  
 offline() Take station offline, possibly temporary  
 shutdown() Shutdown station and all devices associated with it, at game exit.

35 Class powerpit.iioB.IIOBase 666 extends Object. It is a main java interface for  
 access to the physical IIOB or IIOB simulator software, in test. poll() method scans the  
 IIOB for new data or state changes and informs the other devices; translates and  
 synchronizes access to the IIOB. Subclassed by IIOB (for physical i/o) or IIOBSimulator  
 40 (for test exercise of code modules). The class includes:

commitWrite() Commit write buffer to iioB.

configure() Send configuration command to iioB; should return 0.  
 getState() Retrieve iioB state: 0 = iioB is not responding.  
 poll() scan i/o board for new inputs; deliver writes for current cycle of game loop  
 readCoinOut() return the accurate coin-out value  
 5 readLine(boolean, int) Read the integer value of a line for input or output.  
 readPort(int) read current data from serial port  
 readStatus() Return the iioB status (0=okay; 1=really bad).  
 readTrackball() Return trackball deltas in byte[2] array.  
 registerCoinValidator(IioBDeviceInterface, int, int, int, int, int, int) Register  
 10 cc40/cc46 coin validator  
 registerHopper(int, int, IioBDeviceInterface)  
 register hopper lines for motor and sensor  
 registerLED(byte) Register LED for configuration (write-only).  
 registerLine(IOLineConfiguration, boolean, int) Register an input or output line  
 15 for call-back, via IioBDevice::getData()  
 registerPort(PortConfiguration, int) Register a serial port for use.  
 registerTrackball(byte, IioBDeviceInterface) Register trackball by providing  
 update interval in 10 ms units, and owner for notification.  
 reset() Hardware reset for iioB; should return s\_iWasReset.  
 20 resetHopper() reset hopper error values  
 resetPort(int) reset port data after error  
 signature() iioB signature routine - just a shell here; implemented only in IIOB  
 start() Send start command to iioB; should return 0.  
 statusHasChanged() allow hopper to check status byte change bit  
 25 stop() Send shutdown command to iioB; should return 0.  
 writeCoinOut(int) write coins to dispense  
 writeLED(int, byte[], byte[]) Send data to seven-segment display.  
 writeLine(int, int) Write a value to an output line.  
 writePort(Port, int) Write data packet to a serial port (e.g.: bill validator or printer)  
 30

Class powerpit.devices.PlayerStation 669 is a public class. PlayerStation extends  
 Station implements PlayerInterfac. PlayerStation implements the methods for  
 communication with a game and dependent devices for a single-player game such as a  
 video slot. The class includes:

35 allowValidators() allow coin and bill acceptance  
 attendantKey() is the attendant key on?  
 attendantLit() is the service lamp lit?  
 blinkButtons(int[]) blink player buttons of the specified types  
 blinkButtonsEQvalue(int, int) blink player buttons of the specified type only if  
 40 they are equal to the specified value (e.g. blink only the bet 5 button  
 instead of all bet buttons)  
 clearHistory() reset history log only  
 dimButtons(int[]) Dim player buttons of the specified types  
 disable() Disable player station, as for alarm.  
 45 disableCashout() explicitly lock out the cashout button  
 displayLog(int) display a log of the specified type as ASCII text

enable() Enable player station; called by game after meter audit.  
 enableCashout() enable normal cashout function  
 getBet() Return value of bet meter.  
 getCashIn() interface for total cash in from all sources (audit check)  
 5    getCashOut() interface for total cash out from all sources (audit check)  
 getCoinIn() interface for coin-in meter (audit check)  
 getCoinOut() interface for coin-out meter (audit check)  
 getCredit() Returns value of credit meter.  
 getIndex() Return player index  
 10    getPaid() Returns value of paid meter.  
 handpay() clear handpay-required tilt  
 increaseBet(long) Increment bet meter; debit credit meter.  
 inhibitValidators() game loop interface to inhibit credit acceptance  
 lightButtons(int[]) Light player buttons of the specified types  
 15    lightButtonsEQvalue(int, int) light player buttons only if they are equal to the  
       specified value and type; if they are the specified type but not the right  
       value, then turn them off.  
 lightButtonsLEvalue(int, int) light player buttons only if they are less than or  
       equal to the specified value  
 20    offline() Take player station offline.  
 playMessage(PlayMessage) Process play message from game; causes game, bet  
       win meters to increment; logs game outcomes.  
 poll() poll the devices associated with this station  
 processMessage(Message) process messages for this station  
 25    resetLogs() reset any logs that need it  
 shutdown() Shut player station down at game exit.  
 shutdownOnline() shut down the online system in case of severe error  
 undoBet(long) Subtracts amount from bet meter; adds credits to credit meter.

30

Class powerpit.devices.DealerStation 670 is a public class. DealerStation extends  
 Station implements NetworkInterface. is provides a dealer station for game server in  
 multiplayer games. The class includes:

35    connect(Connection) add connection for remote player  
 disable() Disable station, as for alarm.  
 disconnect(Connection) remove connection for remote player  
 enable() Called from GameApplet::start() or BaseGame::start() in powerpit.mains.  
 poll() poll the devices associated with this station  
 40    processMessage(Message) process messages for this station  
 shutdown() shut down dealer and all connected player stations at game exit

Class powerpit.devices.PlayerStub 671 is a public class. PlayerStub extends  
 Object

45    implements PlayerInterface PlayerStub. It is used by the game server for multiplayer

games. It stores the data locally for a remote player station. The class includes:

- disable() Disable player station, as for alarm.
- enable() Enable player station; called by dealer when connected.
- getBet() Return value of bet meter.
- 5 getCredit() Returns value of credit meter.
- getIndex() Return player index
- getPaid() Returns value of paid meter.
- getState() Return player state
- increaseBet(long) Increment bet meter.
- 10 offline() Take player station offline.
- playMessage(PlayMessage) Process play message from game.
- shutdown() Shut player station down at game exit.

Interface powerpit.devices.PlayerInterface 672 is a public interface.

- 15 PlayerInterface extends StateInterface. PlayerInterface defines the methods for communication between players and a game. Implemented by PlayerStation, NetworkPlayerStation and PlayerStub
- getBet() Return value of bet meter. The class includes:

- 20 getCredit() Returns value of credit meter.
- getIndex() Return player index
- getPaid() Returns value of paid meter.
- increaseBet(long) Increment bet meter.

- Class powerpit.jni.ServerSocket 673 is a public class. ServerSocket extends
- 25 Socket and implement Berkeley sockets interface with non-blocking sockets. The class includes:

accept() accept new connection; used by DealerStation on game server.

- Class powerpit.jni.ClientSocket 674 is a public class. ClientSocket extends Socket
- 30 Implement Berkeley sockets interface with non-blocking sockets. It does NOT transmit raw data, only subclasses of Message. Used by Connection, ServerSocket; used indirectly by DealerStation and NetworkPlayerStation. The class includes:

- recv() return Message to caller.
- 35 send(byte[], int) write Message to network.

Class powerpit.devices.ButtonPanel 675 is a public abstract class. ButtonPanel extends Object. It describes the player button panel for a game. Similar to SwitchHarnee, Candelabra and MeterModel, it is subclassed for the machine type. E.g: NVSMButtonPanel describes the buttons for a video slot, while NVBJPButtonPanel

describes the buttons available for a player station in multi-player blackjack. The class includes:

5       addBetButton(int, int, long) used by constructor of subclass to add a bet button  
           addButton(int, int, int) add a button; should only be used by the constructor.  
           addPaylineButton(int, int, long) used by constructor of subclass to add a payline  
               selection button  
           blinkButtons(int[]) blink (and enable) all buttons of the specified types  
           blinkButtonsEQvalue(int, int) blink (and enable) all buttons of the specified type  
               only if their value is equal to the specified value.  
 10       dimButtons(int[]) dim (disable) all buttons of the specified types  
           disable() disable all buttons for alarm; attendant, cashout and show are never  
               disabled  
           enable() re-enable buttons after alarm  
           isLit(int) get state of all buttons of a particular type; currently, this is most useful  
 15       for the attendant button.  
           lightButtons(int[]) light (enable) all buttons of the specified types  
           lightButtonsEQvalue(int, int) light (and enable) all buttons of the specified type  
               only if their value is equal to the specified value.  
           lightButtonsLEvalue(int, int) light (and enable) all buttons of the specified type  
 20       only if their value is less than or equal to the specified value.  
           register(IIOBase) register all the buttons with iioB

Class powerpit.online.OnlineSystem 676 is a public class. OnlineSystem extends  
 Object. This class contains the java part of the common online accounting system  
 25    methods. These methods call the C part of the common interface which in turn  
       communicate with the protocol translator, which runs asynchronously as a separate  
       process. This enables all game software to use a common interface to a family of protocol  
       translators which need not be resident on the machine's main motherboard at all. NOTE:  
       the term "slave" is used to indicate the protocol translation task itself.

30       check() Native method to check to see if the slave task is still running.  
           doStart() check that slave is running; stores pid of "slave" if everything's okay;  
               throws exception otherwise.  
           nativepollRx(byte[]) Native method used to check the message queue to see if the  
               slave task has sent a message to the game.  
 35       poll() process incoming messages.  
           pollRx() This method is used to poll the "slave" for any action or data that it may  
               be sending to the game.  
           qDenom(int, byte) This method calls the native method qTx to put an event that  
               requires a denomination code into the queue for the slave.  
 40       qEvent(int) This method calls the native method qTx to put an event (with no  
               data) into the queue for the slave.  
           qHandpayWin(long, long, byte) This method calls the native method qTx to put a  
               "handpay won" or "handpay collected" event into the queue for the slave.

qID(byte[]) This method calls the native method qTx to put a "set machine id" event into the queue for the slave.  
 qMeter(int, long) This method calls the native method qTx to put an event that requires an amount for a meter value into the queue for the slave.  
 5 qOptions(short) This method calls the native method qTx to put a "set options" event into the queue for the slave.  
 qPayBackPercent(String) This method calls the native method qTx to put a "set payback percent" event into the queue for the slave.  
 qProgressiveGroup(byte) This method calls the native method qTx to put a "set progressive group" event into the queue for the slave.  
 10 qProgressiveMeter(byte, long) This method calls the native method qTx to put a "adjust progressive meter" event into the queue for the slave.  
 qReadMeter(char, int) This method calls the native method qTx to put a "read meter" event into the queue for the slave.  
 15 qReelID(byte[]) This method calls the native method qTx to put a "set reel strip id" event into the queue for the slave.  
 qTx(int, byte[]) Native method that actually put a message from the game to the slave into the named pipe.  
 quit() Native method to shut down the slave task.  
 20 qWriteMeter(char, int, long) This method calls the native method qTx to put a "write meter" event into the queue for the slave.  
 resume() Native method used to try to resume a slave task that has been suspended.  
 shutdown() tell slave to clean up and exit  
 25 sizepollRx() Native method which simply returns the size of the message buffer needed by the nativepollRx method.  
 start(Meter[], String, String, String, int, int) This method checks that the slave has started; sends alarm to owner if something's wrong.  
 suspend() Native method used to send a message to the slave task requesting it to suspend itself.  
 30 translate(Message) This subroutine is supposed to be called by PlayerStation.processMessage() to translate messages that the player station gets into stuff that the online accounting system understands.  
 trigger() Call this native method after you have queued all desired events (or commands) to the slave task. Wakes up slave to process a group of data messages atomically.  
 35

Class powerpit.devices.NetworkPlayerStation 677 is a public class in which NetworkPlayerStation extends PlayerStation. The class implements NetworkInterface  
 40 NetworkPlayerStation implements the methods for communication with a remote game server, in addition to basic player functionality. The class includes:  
 connect(Connection) connect to game server  
 disable() Disable player station, as for alarm.  
 disconnect(Connection)  
 45 enable() Enable player station; called by game after meter audit.  
 offline() Take player station offline.

poll() poll for inputs and process output to devices  
 processMessage(Message) process messages from devices, game server or local  
 game loop  
 sendDelta(int, long) send meter deltas to the dealer  
 shutdown() Shut player station down at game exit.

5

Class powerpit.devices.Connection 678 is a public class. Connection extends  
 Device Service player station connection. Same class is used by client and server.  
 Messages are sent using the send() method. Messages are received by the poll() method.  
 If no messages are received for 15 seconds the connection will be dropped. The class  
 includes:

10

getPlayerIndex() Retrieve index of player.  
 nextMessage() Receive network message.  
 poll() poll is called each time through the game loop to scan for input  
 send(Message) Send network message.  
 shutdown() terminate network connection

15

Class powerpit.devices.Device 680 is a public abstract class Device extends  
 Object implements GameInterface which is an abstract base class for all devices. Devices  
 are strictly event driven, using the poll() method invoked by the game loop; there is no  
 monitoring Thread in any device.

20

getParameter(String) Get parameters from the configuration file.  
 getType() Return device subtype; -1 indicates that no subtypes are defined.  
 processMessage(Message) The method that gets called to distribute messages.

25

Fig 6d illustrates the relationships between the Java classes associated with a  
 Device Layer Context. The Device Layer Context comprises a Device 690 that interacts  
 with GameInterface 691, LiobDevice 692, and Meter 694. LiobDevice 692 interacts with  
 class liobDeviceInterface 693 which in turn interacts with HardMeter 695. HardMeter  
 695 interacts with classes CoinOutMeter 697 and Meter 694. CoinOutMeter 697 interacts  
 with classes Hopper 698 and MessageInterface 697, which also interacts with Hopper  
 698. Hopper 698 also interacts with class liobDevice 699. ButtonPanel 6011 interacts  
 with class PlayerButton 6012. Candleabra 6021 interacts with class Candle 6022.  
 SwitchHarness 6031 interacts with class Switch 6032. MeterModel 6041 interacts with  
 class Meter 6042. LiobDevice 6001 ineracts with classes PlayerButton 6008, Candle  
 6007, Switch 6006, Printer 6005, LED 6004, Trackball 6003, and CreditDevice 6002.  
 CreditDevice 6002 interacts with classes CoinValidator 6010 and BillValidator 6009.

30

35

Class powerpit.devices.Meter 694 is a public class. Meter extends Device Base class for all meters. Implements functions for soft meter. All defined types are included  
 5 in the MeterModel for all jurisdictions. The class includes:

decrement(long) Decrement by specified amount.  
 getUnrolled() Return unrolled value of meter - always 0 unless it's a hard meter  
 getValue() Return value of meter.  
 increment(long) Increment by specified amount.  
 10 reset(long) Basic reset method.  
 typeString(int) translator for known meter types - integer values are defined for all  
 meter types required in Nevada and GLI U.S. jurisdictions, including door  
 types and individual bill denominations from 1 to 1 million.

15 Class powerpit.devices.HardMeter 695 is a public class. HardMeter extends Meter implements IiobDeviceInterface and is derived from base class Meter. It encapsulates player station elector-mechanical meter. Note presence of attribute IUnrolled; the amount not yet sent to the physical meter. Note that all the functionality from IiobDevice must be implemented here, since HardMeter is not a subclass of IiobDevice decrement (long)

20 Overrides base class. The class includes:

disable() Disable the meter - stops sending pulses to the iioB.  
 enable() Invoked by the IIOB when data transfer is ok.  
 getData() Retrieve changed data for this meter (meter has rolled 1 count)  
 increment(long) Overrides base class - causes meter to start rolling if enabled.  
 25 register(IIOBBase) Cause the hard meter to register with the IIOB.  
 reset(long) Overrides base class - resets unrolled amount to 0.

Class powerpit.devices.CoinOutMeter 697 is a public class. CoinOutMeter extends HardMeter implements MessageInterface and is derived from base class  
 30 HardMeter. It encapsulates physical coin-out meter relationship with hopper. The class includes:

cashing() let meter model return cashing flag based on hopper and meter  
 disable() Disable the meter  
 enable() Invoked by the IIOB when data transfer is ok.  
 35 increment(long) Overrides base class; send coin-out to hopper  
 processMessage(Message) The method that gets called to distribute messages.  
 register(IIOBBase) Cause the hard meter to register with the IIOB.  
 resetTilts() reset hopper tilt flags so messages will be sent once more for hopper-  
 related tilts  
 40 rollCoinOut() hopper calls this routine to roll meter

Class `powerpit.devices.Hopper 698` is a public class. `Hopper` extends `IiobDevice`. `Hopper` Device controlled by coin-out meter to physically dispense coins. The class includes:

- 5        `coinOut(int)` called by coin-out meter to indicate number of coins to dispense
- `disable()` Invoked by IIOB when data transfer is not ok, or by coin-out meter
- `enable()` Invoked by the IIOB when data transfer is ok, or by coin-out meter.
- `getData()` Retrieve changed coin-out data for hopper
- `processMessage(Message)` The method that gets called to interpret iio hopper
- 10        status.
- `register(IIOBase)` Cause the hard meter to register with the IIOB. - called by
- coin-out meter.
- `resetTilts()` allow tilt message to generate.
- `throttle(AlarmMessage)` inhibit duplicate tilt messages until `resetTilts()` is called.

15        Class `powerpit.devices.CreditDevice 6002` is a public abstract class. `CreditDevice` extends `IiobDevice` and is an abstract base class for monetary input devices. It is derived from base class `IiobDevice`; encapsulates player station credit device; e.g.: eft, coin acceptor or bill acceptor. Note that `BillValidator`, `CoinValidator` and their subclasses

20        implement the allow/inhibit interfaces to preserve state through iio resets so a validator that was off because a limit was reached does not come on, or vice versa. The class includes:

- `allow()` player station calls this to allow acceptance of credit inputs by this device
- `inhibit()` player station calls this to inhibit acceptance
- 25        `resetInCount()` player station calls this at cashout or handle-pull to reset amount in
- since last game or cashout.

Class `powerpit.devices.Trackball 6003` is a public class. `Trackball` extends `IiobDevice`. `Trackball` device. Note that other pointing devices would be similar, but

30        might not be subclasses of `IiobDevice`, since touchscreens or mice could be handled by the mouse port on the motherboard. The class includes:

- `getData()` Retrieve changed data for this device
- `register(IIOBase)` Cause the trackball to register with the IIOB.

35        Class `powerpit.devices.LED 6004` is a public class. `LED` extends `IiobDevice` and controls the output to seven-segment display with 8 characters. The class includes:

- `getData()` empty method to satisfy abstraction in `IiobDevice`.
- `register(IIOBase)` Cause the LED to register with the IIOB.

Class powerpit.devices.Printer 6005 is a public class. Printer extends IiobDevice  
Printer device. It is controlled by player station. Note: this is the interface for a serial  
printer connected via the i/o board. A parallel printer connected to the motherboard  
would be functionally equivalent, but would act through the parallel port driver rather  
5 than the iioB interfaces. The class includes:

getData() Retrieve changed data for this device; e.g. ack, device not ready, etc.  
print(String) Print something  
register(IIOBBase) Cause the printer to register with the IIOB.

10 Class powerpit.devices.Switch 6006 is a public class. Switch extends IiobDevice  
and monitors a single security switch to send alarm for state change. There are many of  
these, organized within a SwitchHarness. Note: all switches are iioB inputs; they cannot  
be turned on and off by the station. The class includes:

alarmOff() return value of alarm off  
15 alarmOn() return value of alarm on  
enable() Invoked by the IIOB when data transfer is ok.  
getData() Retrieve changed data for this switch  
isOn() Returns value of fOn.  
isReversed() return true if switch is "reversed" polarity; i.e. if switch is active  
20 high.  
register(IIOBBase) Cause the switch to register with the IIOB.

Class powerpit.devices.Candle 6007 is a public class. Candle extends IiobDevice.  
A single candle on a station; may have 2, 3 or more of these, contained by a subclass of  
25 Candelabra. The class includes:

blink() blink the candle  
getData() Retrieve changed data for this device - required in case of reset  
isBlinking() is it blinking?  
30 isOn() is it on steady?  
off() turn candle off  
on() turn candle on steady  
register(IIOBBase) register candle line with iioB

Class powerpit.devices.PlayerButton 6008 is a public class. PlayerButton extends  
35 IiobDevice and is a single layer button attached to iioB. There are many of these,  
contained in a subclass of ButtonPanel. The class includes:

blinkButton() Blink (and enable) this button  
dimButton() Dim (disable) this button  
40 getData() Retrieve changed data for this device  
getValue() Return fixed numerical value of (bet or playline) button.

isLit() Return true if button is lit (or blinking), false if dim.  
 lightButton() Light (enable) this button  
 register(IIOBase) Cause the hard meter to register with the IIOB.  
 sendMessage() Send message - used by multiplayer games for attendant button  
 5     typeString() This instance method returns a string containing a button's type as  
        well as its value (for bet and payline buttons).  
 typeString(int) This static method returns a string describing what a button type is.

Class powerpit.devices.BillValidator 6009 is a public abstract class. BillValidator  
 10   extends CreditDevice implements MessageInterface which is a serial bill validator  
 abstract base class. It uses serial.c in .../libLinux . The class includes:

allow() allow acceptance, if enabled.  
 disable() Invoked by IIOB when data transfer is not ok; should be called by  
           subclass after device-specific messages.  
 15   enable() Invoked by the IIOB when data transfer is ok.  
 inhibit() inhibit acceptance without disabling.  
 init() handle initialization sequence  
 poll() troll for responses; check timeouts.  
 processMessage(Message) The method that gets called to distribute messages.  
 20   register(IIOBase) Cause the bill validator to register with the IIOB.  
 resetInCount() reset bill count after handle-pull or cashout  
 sendAlarm(int) send tilt; specific to derived type.

25     Class powerpit.devices.CoinValidator 6010 is a public abstract class.  
 CoinValidator extends CreditDevice CC-40 coin validator. Notify coin-in meter via  
 CreditMessage to station. Note: there are numerout subclasses implemented depending  
 on diverter and hopper types. The class includes:

allow() same as enable for cc-40 and cc-46  
 30   disable() invoked by IIOB when data transfer is not ok.  
 enable() invoked by the IIOB when data transfer is ok.  
 getData() check coin-validator on tilt or coin-in  
 inhibit() same as disable for cc-40 and cc-46  
 poll() poll for severe errors even when disabled  
 35   register(IIOBase) Cause the device to register with the IIOB.  
 resetInCount() reset coin-in count after cashout or handle-pull  
 setOff() turn validator off and clear flag  
 setOn() turn validator on and set flag

40     Below are all the parameters understood by the basic game software. Most games  
 would require various additional parameters for control of game-specific features such as  
 sound, video or special device options.

The name of applet to run for game (required for graphical display)  
**GAME\_APPLET** powerpit.docs.SlotMachine.

The dimensions for applet display (required for graphical display)  
**WIDTH** = 640 and **HEIGHT** = 480.

5 The game denomination: cents per credit (required) is **DENOMINATION**  
 = 100.

The station type (required) is **STATION** powerpit.devices.PlayerStation,  
**STATION\_CLASSES** powerpit.mains.BaseGameInterface, and  
**STATION\_VALUES** owner.

10 The image to display in background (required for graphical display) is  
**PLAYER\_BACKGROUND** blackjack/images/Bjdback.gif.

The pure java random number generator to use (required for dealer or  
 single player) **GENERATOR** powerpit.chance.JKnuthV2Random. LSR with  
 saved seed (seed is not actually saved without PROMDisk installed)

15 **GENERATOR** powerpit.chance.KnuthWithRNG. **GENERATOR\_CLASSES**  
 java.lang.Integer **GENERATOR\_VALUES** =200.

The candles (optional) is **CANDLE** powerpit.docs.NVSMCandles,  
**CANDLE\_CLASSES** powerpit.devices.PlayerStation, **CANDLE\_VALUES**  
 owner.

20 The online system (comment this out to test bill validator on PC)  
 (optional) is **ONLINE** powerpit.online.OnlineSystem, **ONLINE\_CLASSES**  
 powerpit.devices.PlayerStation, **ONLINE\_VALUES** owner, **COMPANY** "IGCA  
 ", **PAYBACK** 96.84, and **GAME\_ID** "NVSSF".

The I/O board (required) is **IIOB** powerpit.iio.IIOBSimulator, **IIOB**  
 25 powerpit.devices.IIOBDummy, **IIOB** powerpit.iio.IIOB, **IIOB\_CLASSES**  
 powerpit.devices.Station, and **IIOB\_VALUES** owner.

The meters (required) is **METER\_MODEL**  
 powerpit.docs.NVSMMeterModel having **METER\_MODEL\_CLASSES**  
 powerpit.devices.Station, and **METER\_MODEL\_VALUES** owner.

30 The Logs: optional, but only one of each type and are disk-based meter log  
 for recovery: **METER\_LOG** powerpit.mains.audit.JMeterLog,

METER\_LOG\_CLASSES java.lang.String, and METER\_LOG\_VALUES  
/tmp/test\_dmeter\_log.

The disk-based event log for alarms are EVENT\_LOG  
powerpit.mains.audit.JEventLog EVENT\_LOG\_CLASSES java.lang.String  
5 java.lang.Integer, and EVENT\_LOG\_VALUES /tmp/test\_devent\_log 100.

The disk-based state log for recovery is STATE\_LOG  
powerpit.mains.audit.JStateLog, STATE\_LOG\_CLASSES java.lang.String, and  
STATE\_LOG\_VALUES /tmp/test\_state\_log.

The disk-based history log for play steps is PLAY\_LOG  
10 powerpit.mains.audit.JPlayLog, PLAY\_LOG\_CLASSES java.lang.String, and  
PLAY\_LOG\_VALUES /tmp/test\_play\_log.

The NVRAM meter log for recovery (optional) is METER\_LOG  
powerpit.mains.nvram.MeterLog, METER\_LOG\_CLASSES java.lang.Integer,  
and METER\_LOG\_VALUES = 5.

15 The NVRAM event log for alarms (optional) is EVENT\_LOG  
powerpit.mains.nvram.EventLog, EVENT\_LOG\_CLASSES java.lang.Integer  
java.lang.Integer, EVENT\_LOG\_VALUES = 5 100. The NVRAM state log for  
recovery (optional) is STATE\_LOG powerpit.mains.nvram.StateLog  
STATE\_LOG\_CLASSES powerpit.mains.BaseGameInterface java.lang.Integer,  
20 and STATE\_LOG\_VALUES owner = 5.

The NVRAM history log for play steps (optional) is PLAY\_LOG  
powerpit.mains.nvram.PlayLog PLAY\_LOG\_CLASSES java.lang.Integer  
java.lang.Integer, and PLAY\_LOG\_VALUES = 5 25.

The switches (optional) is SWITCH\_HARNESS  
25 powerpit.docs.NVSMSSwitchHarness, SWITCH\_HARNESS\_CLASSES  
powerpit.devices.Station, and SWITCH\_HARNESS\_VALUES = owner.

The is button panel (optional) BUTTONS  
powerpit.docs.NVSMButtonPanel, BUTTONS\_CLASSES  
powerpit.devices.PlayerStation java.lang.Boolean, and BUTTONS\_VALUES  
30 owner true.

The bill validator (optional) is BILL\_VALIDATOR  
powerpit.devices.CBVValidator, BILL\_VALIDATOR

powerpit.devices.JCMValidator, and BILL\_VALIDATOR\_CLASSES  
powerpit.devices.PlayerStation java.lang.Integer.

The bill validator uses serial port 0 on the iioB is  
BILL\_VALIDATOR\_VALUES owner 0.

5 The test values for direct PC serial port hookup with the bill validator is  
BILL\_VALIDATOR\_CLASSES powerpit.devices.PlayerStation java.lang.String,  
and BILL\_VALIDATOR\_VALUES owner "/dev/cua0".

The coin validator (optional) is COIN\_VALIDATOR  
powerpit.devices.BJCoinValidator, COIN\_VALIDATOR  
10 powerpit.devices.ColeCoinValidator, and COIN\_VALIDATOR\_CLASSES  
powerpit.devices.PlayerStation.

The ColeCoinValidator with 1 diverter is java.lang.Integer  
java.lang.Integer java.lang.Integer, BJCoinValidator with 2 diverters  
java.lang.Integer java.lang.Integer java.lang.Integer, java.lang.Integer  
15 java.lang.Integer, BJCoinValidator with 2 diverters, coin-in, tilt, drop gate, hopper  
gate, hopper full, power, COIN\_VALIDATOR\_VALUES owner = 5 24 0 6 9 4,  
ColeCoinValidator with 1 diverter, coin-in, tilt, drop gate, hopper full, power, and  
COIN\_VALIDATOR\_VALUES owner = 5 24 6 9 4.

The hopper (optional) is HOPPER powerpit.devices.Hopper,  
20 HOPPER\_CLASSES powerpit.devices.CoinOutMeter, java.lang.Integer and  
java.lang.Integer.

The motor line, sensor line is HOPPER\_VALUES owner = 7 8.

The printer (optional) is PRINTER powerpit.devices.Printer,  
PRINTER\_CLASSES powerpit.devices.PlayerStation java.lang.Integer, 1 is  
25 assigned serial port on I/O board, PRINTER\_VALUES owner = 1.

Optional parameters for which defaults are supplied tilt when player wins  
this many dollars in credits (clear with key), defaults to 10000, and  
JACKPOT\_LIMIT = 1000.

Turn off coin validator when this many dollars are accepted between  
30 games and defaults to unlimited, COIN\_LIMIT = 1500; turn off bill validator  
when this many dollars are accepted between games and defaults to unlimited,  
BILL\_LIMIT = 1500; turn off eft transfers in when this many dollars are accepted

between games and defaults to unlimited, EFT\_LIMIT = 3000; hopper limit (in dollars) and defaults to 1200 (clear with key), HANDPAY = 1200; minimum bet (in credits) and defaults to 1, MIN\_BET = 1; maximum bet (in credits), may not exceed 100 times minimum, MAX\_BET = 100; stop accepting credits (in dollars), defaults to 3000, VALIDATORS\_OFF = 3000; game controls when the error candle goes off - any value for this parameter allows game-control of the error candle, MANUAL\_CANDLE = true; unreasonable win amount strictly greater than any possible jackpot (credits) defaults to 999,999,999, WIN\_LIMIT = 1000000; short-circuit version test (optional for test of RNG without NVRAM only) (any (non-blank) value at all), NO\_VERSION = true; tournament mode; implies NO\_VERSION (any (non-blank) value at all), TOURNAMENT = true; tournament round in minutes - can be reset manually and defaults to 15, TOURNAMENT\_MINUTES = 15; and tournament credits per round - can be reset manually and defaults to 1000, TOURNAMENT\_CREDITS = 1000.

A set of game-specific parameters are discussed below. For multi-player games, the only additional parameters required are for the dealer station. In most cases however, the dealer would not require any devices other than meters, the I/O board and possibly candles.

Fig. 7 illustrates the additional classes and relationships for multi-player games. Fig. 7 also illustrates the relationships between the Java classes associated with Multi-Player Classes. The Multi-Player classes comprise a DealerGame 701 that interacts with classes JRandom 702, PlayerStub 703 and DealerStation 704. DealerStation 704 interacts with classes P-layerStub 703, Connection 711, and ServerSocket 710, which in turn interacts with classes Connection 711 and ClientSocket 712. Connection 711 also interacts with class ClientSocket 712 as well as ClientSocket 723 and NetworkPlayerStation 720. NetworkPlayerStation 720 itself interacts with ClientSocket 723 OnlineSystem 722, and PlayerGame 721.

Dealer Game 701 is an abstraction to indicate the portion of a multi-player game which contains a game loop based on either BaseGame (610) or GameApplet (640) and references a DealerStation (670) with PlayerStubs (671).

Player Game 721 is an abstraction to indicate a single, remote player in a multiplayer game.

Parameters listed below that differ from the list above that are required for any dealer.

5                   The station type (required) STATION powerpit.devices.DealerStation, STATION\_CLASSES powerpit.mains.BaseGameInterface, and STATION\_VALUES owner.

                  The player stations to load (required for dealer) uses PLAYERS = 5.

                  The dealer image to display in background (required for graphical display)  
10 DEALER\_BACKGROUND blackjack/images/Bjdback.gif.

The only change in configuration for a remote player in a multi-player game is the station type. A remote player would have all the devices a single player is likely to have.

                  The station type (required) is STATION  
15 powerpit.devices.NetworkPlayerStation, STATION\_CLASSES powerpit.mains.BaseGameInterface, and STATION\_VALUES owner.

Each station is responsible for starting attached peripheral devices in the correct sequence both at startup and when re-enabled after an alarm condition (tilt) is cleared. This is handled by low-level software that is not game-specific. In addition, a dealer  
20 station starts the server network connection, when the game loop becomes active. A remote player station attempts to connect to the remote server and loads, but does not start its attached devices until the network connection is made.

The online accounting system may be of any industry supported type of online system; e.g.: SDS, SAS, CDS, Daycom, etc. The online system is started when a station  
25 is first enabled and remains active until shutdown in order to report alarms. The OAS runs as a separate process at a high priority, so it can respond to host messages in real-time. The I/O board is configured as the supported peripheral software is loaded. It is started only when all other items are loaded and in the case of a player in a multi-player game, only after the player is connected on the network. The I/O board remains active  
30 during alarm conditions, except at shutdown or when the alarm is caused by a failure of

the I/O board. If the I/O board fails to respond correctly to regular system polls at any time, a reset operation is performed and an alarm declared. If the I/O board fails to respond to the reset, the alarm may only be cleared by re-booting; such a failure most likely indicates a hardware defect. The technical details of the I/O board design and related functionality may be found within co-pending U.S. Patent Application entitled

5 SYSTEM AND METHOD FOR PROVIDING A REAL-TIME PROGRAMMABLE INTERFACE TO A GENERAL-PURPOSE NON-REAL-TIME COMPUTING SYSTEM, Serial No. 09/395,647 which is concurrently filed with the patent application and incorporated by reference herein in its entirety.

10 The primary duty common to all game loops is to continually call a poll() routine, implemented in both BaseGame and GameApplet base classes, which polls the station software for device messages including alarms and in the case of a multi-player game for network messages from the remote partners. Whenever poll() is called, it should be followed by a call to nextMessage(), which retrieves the polled data in a standard system

15 message packet, derived from powerpit.messages.Message.

In the case of an audit check event, the meters must be cleared by entering audit mode, and opening the logic cage; this causes a RAM\_CLEAR event to be logged. An audit check event occurs whenever credit is not equal to the sum of cash-in plus cumulative win minus cash-out, minus cumulative bet, minus current bet. Current bet is

20 added to cumulative bet at the end of each game; the audit check is performed at the beginning of the game and each time a bet is placed.

RAM corruption or any change in the configuration causes a RAM/Version check event to occur. In the case of a single player game (e.g.: Storybook Fantasy), or player station for a multi-player game (e.g.: BJ Blitz Blackjack) the event is cleared as with the

25 audit check event above. For the dealer station in a multi-player game, the alarm is cleared by opening the logic door.

An audit-check tilt may only be cleared by turning the attendant key to its "on" position and opening the logic door to clear the meters; a ram-clear event is logged at this time. On multi-player dealer stations, the attendant key is not required; opening the logic

30 cage clears RAM and restores game function.

A hopper, bill validator or IO board tilt of any kind stops the game as with an open door. These tilts may only be cleared by opening at least one door and then closing all doors.

- Any door open results in a tilt as well as hand-pay-required or a failed audit check.
- 5 In multi-player games, a dealer tilt will disable all player stations as well. A power-down alarm results in the shutdown of the station that declared it. In multi-player games, a dealer power-down will result in a shutdown by the player stations as well.

- On graphical player stations, if no hand-pay alarm is in force, turning the attendant key to the "on" position causes the station to be disabled as for an alarm. Pressing the
- 10 cashout key cycles from one data set to the next; pressing the attendant key pages through the current data. Opening the logic door clears the current log and in the case of clearing the meters enters a "Ram cleared" alarm in the event log. Clearing the meters in audit mode is the only way for game play to proceed after an audit check alarm is raised. Data is displayed or cleared by `powerpit.mains.audit.Audit`; an instance of this class is owned
- 15 by the game applet on player stations (only). Audit is invoked by `GameApplet::audit()`.

RAM based meters are updated immediately as credits are transferred, and the resulting values are logged. Electromechanical meters are updated asynchronously, until all deltas have been added; the "unrolled" amount is logged each time the meter is incremented for power-fail recovery.

- 20 In the case of game-play, credits are transferred to an interim bet meter rather than directly to the cumulative bet (Nevada coin-in) meter. At the end of the game cycle, these credits are returned to the credit meter in the event of a tilt, or then added to the cumulative bet meter. At this time, credits paid by the game are also added to the cumulative won meter.

- 25 Specific message classes are provided for:

1. Remote player connection; initializes credit and bet values on the dealer station and marks the player as enabled.
2. Play step messages; these are sent from the game to the player for logging and in the case of multi-player games, to control the game loop on the remote station.

3. Meter messages; these generally contain full meter values for logging or synchronization of remote values in multi-player games. In multi-player games, deltas are also sent by players to the dealer to update cumulative dealer meters.

4. Credit messages are sent by credit devices such as coin validators, bill validators or eft systems to the player station software. They contain fields indicating source of the credits, denomination in the case of bills, and whether the credits should be added to the drop meter or not.

5. Button messages are sent to the player station by the button devices; in single-player games, they are made available to the local game loop; in multi-player games they are sent to the dealer to relay to the remote game server.

6. Alarm messages are sent only to the local game loop. Based on the outcome, at the game level, enable or disable commands may be sent by the player to the dealer (which only affect the status in that player's database), or by the dealer to the player. Alarms for hand-pays and cash-outs contain the amount of credits involved and the basic alarm identifier.

7. Trackball messages are sent (via the player station software) to the local game loop to affect the position of a pointer.

8. Command messages are sent to enable or disable players or shut down the system in an orderly fashion when an error occurs.

9. An ACK message type is provided to indicate completion of some lengthy action (e.g.: animation) by a remote player in multi-player games.

10. A Keep-alive message type is provided to verify communication links between the dealer and remote players in multi-player games.

All messages are subclasses of `powerpit.messages.Message`. They include certain required methods (functions, subroutines, whatever):

```
public static final byte[] getClassName (byte[] data)
```

This method is used to recover the class name of a message in packed binary form, for reconstruction by the methods of `Class`, `Object` and the classes in package `java.lang.reflect`.

.../powerpit/devices/Connection.java in the source tree has a good example of this in its `nextMessage()` method.

Public byte[] pack () - This method stores the data from the message in a packed binary format suitable for network transmission or logging.

.../powerpit/devices/Connection.java in the source tree has a good example of this in its nextMessage() method.

5       Public int unpack (byte[] data)- This method decodes the packed data returned from a log or network transmission. .../powerpit/devices/Connection.java in the source tree has a good example of this in its send() method.

Public static final String date (long ldate) - This method returns a string of the form mm/dd/yy hh:mm:ss from the long integer stored in each message.

10       Public String[] display () - This method is required to display the data stored by the message; used by audit - particularly important and tedious to implement in the play step messages.

Fig. 8 illustrates the relationships among the various classes employed in messaging. The Message Classes comprise a Message 801 that interacts with classes  
15    CommandMessage 802, AlarmMessage 803, PlayMessage 804 (which in turn interacts with classes InitialStep 816, IntermediateStep 815, and FinalStep 814), CreditMessage 805, MeterMessage 806, ConnectMessage 807, TrackBallMessage 808, ButtonMessage 809, KeepAlive 810, Ack 811, and RandomDraw 813. PlayMessage 804 interacts with classes InitialStep 816, IntermediateStep 815, and FinalStep 814. AlarmMessage 803  
20    interacts with class AlarmWithValue 812.

Class powerpit.messages.PlayMessage 804 is a public abstract class PlayMessage extends Message and is sent by the game to the player for logging. It is used by remote player station game loop for changing state, redrawing screen, playing sounds, etc.

25       Class powerpit.messages.CreditMessage 805 is a public class. CreditMessage extends Message and is sent by bill-validator, coin validator or online system to add credit.

Class powerpit.messages.MeterMessage is a public class. MeterMessage extends Message Sent by the game to the player for meter updates during the betting phase, or to reflect cancelled play. Sent by a remote player to update the credit values for the game  
30    when credit is added or subtracted by coin-in, bill-in, online system or cashout. It is used by meter logging to store meter values for recovery.

Class `powerpit.messages.ConnectMessage` 807 is a public class `ConnectMessage` extends `Message` . It is sent by remote player to update game meters and notify dealer of new connection.

- Class `powerpit.messages.TrackballMessage` 808 is a public class.  
 5 `TrackballMessage` extends `Message` to notify game of new trackball position.

Class `powerpit.messages.ButtonMessage` 809 is a public class. `ButtonMessage` extends `Message` and is sent by player to indicate a press of a button on the player button panel.

- Class `powerpit.messages.KeepAlive` 810 is a public class. `KeepAlive` extends  
 10 `Message` and is a network watchdog message - no content; just keeps connection active.

Class `powerpit.messages.Ack` 811 is a public class. `Ack` extends `Message` and is sent by a remote player to let the dealer know when a lengthy process is done.

- Class `powerpit.messages.AlarmWithValue` 812 is a public class.  
`AlarmWithValue` extends `AlarmMessage` and is sent by player or dealer to indicate credit  
 15 value involved in alarm for handpay, cashout, jackpot or invalid win. Decode specific log type for RAM corruption.

Class `powerpit.messages.RandomDraw` 813 is a public class. `RandomDraw` extends `Message` and is sent by test console to game machine running serial test software, to indicate random numbers drawn on console.

- Class `powerpit.messages.FinalStep` 814 is a public class. `FinalStep` extends  
 20 `PlayMessage` and is an end of game notification sent by game to player for logging, recalculation of credits, various game meters. Subclassed by individual game.

- Class `powerpit.messages.IntermediateStep` 815 is a public class. `IntermediateStep` extends `PlayMessage` and `Intermediate` play steps sent between dealer and remote player  
 25 in multiplayer games (e.g. blackjack).

Class `powerpit.messages.InitialStep` 816 is a public class `InitialStep` extends `PlayMessage`. Initial play steps such as start of game, etc. primarily sent from

game/dealer to a player. No final bets have been placed when these messages are sent, and generally a game be cancelled, or resumed without penalty at this point.

In the single player game, all player messages are sent only to the game. The base class method `processMessage()` strains out everything except player button presses and trackball messages, which are available to the game loop through its `nextMessage()` method. Alarms (e.g. attendant key on/off) are examined via the `lastAlarm()` and `isAlarmed()` methods in `BaseGame` and `GameApplet`. The player station does not echo play step messages to the game after processing them.

Fig 9 illustrates the relationships between the Java classes associated with Local Message Passing.

In the multi-player game, player messages are relayed to the dealer. The dealer station game loop initiates all play step messages which are then relayed back to the game loop on the relevant player station. Button presses are generally available only to the dealer station game; except for auditing or tournament modes when the cashout, attendant and possibly other buttons are made active in a special way.

Fig 10 illustrates the relationships between the Java classes associated with Network Message Passing.

Logs are available in two flavors. Logs using file-based I/O, primarily for test and development, may be found in package `powerpit.mains.audit`. Logs using binary I/O to NVRAM are in package `powerpit.mains.nvram`.

All logs contain the following interfaces:

```

/** When NMI() is invoked, all logs know it. */
    public static final synchronized void NMI()

/** used by online system in particular */
    public static final synchronized boolean NMIReceived()

/** Abstract method must be implemented by each subclass. Should be implemented
as synchronized, so save operations complete before NMI() can be invoked. */
    public abstract void save(Object object)throws IOException;

/** Recover operation for specific object type to be logged. Should be implemented
as synchronized so save() operation cannot change data while it is being recovered. */
    public abstract Object recover()throws IOException;
```

/\*\* \* Display stored data. Data should be recovered with the \* recover() method, internally, to avoid displaying corrupt data. \*/

public abstract String[][] display() throws Exception;

Each player must log events and play history for auditing. Meters are likewise  
5 required for auditing and also for player station recovery after power-down or other serious error conditions requiring reboot.

Audits are always an instance of powerpit.mains.audit.JEventLog or powerpit.mains.nvram.EventLog. A configurable number (default 100) of events are displayed. All system events are logged by the player, whether they cause an alarm  
10 condition or not.

Meters is an instance of powerpit.mains.audit.JMeterLog or powerpit.mains.nvram.MeterLog. All meters are logged by the player, each time they change value and recovered at power-up.

Play History is an instance of powerpit.mains.audit.JPlayLog or  
15 powerpit.mains.nvram.PlayLog. Initial (handle-pull) and final (game-over) play steps are logged by the player. A configurable number (default 25) of play steps are displayed.

Bills is an instance of powerpit.mains.audit.JBillLog or powerpit.mains.nvram.BillLog. All bills accepted by a player station are logged. A configurable number (default 100) of bills are displayed.

20 Game Logs are configured into most games although they may not be required. Currently, there is no audit mode available for viewing logs on a dealer station.

State Log are an instance of a subclass of powerpit.mains.audit.JStateLog or powerpit.mains.nvram.StateLog. If configured, it is used to recover game state after a power-down or other serious error condition requiring reboot.

25 The Online Accounting System runs as a separate, real-time process. It is initiated by a player station, using an instance of powerpit.online.OnlineSystem. The method powerpit.online.OnlineSystem::translate() is used to transform system messages into online system packets and transfer them. The method powerpit.online.OnlineSystem::poll() is used by the system to acquire online commands  
30 and credit transfers.

When tournament mode is activated, via the TOURNAMENT parameter in the configuration file, several changes occur in the execution of the normal game loop. Play

stops whenever the player's credits go to 0 or the tournament round timer expires. Play resumes when an attendant uses his key to reset the tournament counters and timer (instructions appear onscreen). Hard meters, the hopper and coin and bill validators are all de-activated, whether configured or not, during tournament mode. Meter values are saved in a different area of NVRAM from the normal meters and the RAM version check is disabled. This allows the casino to use the same machines for tournament play as those which run the "real" game, without having to clear the RAM, as when a game version changes for normal play. The online system, if present, will continue to monitor the tournament machines normally.

At shutdown, the game invokes the destroy() method. This causes a flag to be set, indicating that shutdown is in progress, and launches a Thread that is an instance of powerpit.mains.Stopper to complete shutdown processing. This Thread stops the running game loop and issues a shutdown command to the local station. If this is a dealer station, the shutdown command is relayed to the remote player stations.

The I/O board sets a status when it senses that power is failing. This is relayed to the station software, which initiates the shutdown sequence, above, by passing the power-down alarm to the game. Once status bit is set, the online system (if any) is aborted and all logs are closed for further new write operations. Any Java Exception caught in the game loop may cause the system to exit; except in the most severe cases of hardware errors, the meters and other logs in non-volatile RAM will still be preserved in a consistent state.

Except in the case of communications errors between the dealer and individual player stations in multi-player games, it is best to design the game loop to exit upon receiving the first exception before NVRAM may be corrupted or an unrecoverable condition obscures the original error.

Various embodiments of individual game personalities associated with individual games that have been constructed using the gaming device 100. These game personalities comprise a storybook fantasy game, a monster money game, and a blackjack game.

#### Storybook Fantasy Personality

All files required to give the Storybook Fantasy Video Slot "personality" to the basic system described previously, reside in .../powerpit/slots and its subdirectories,

princess/, audio/ and princess/images/. Sounds are stored in audio/ and graphics for the reels and backgrounds are in princess/images/.

Several configuration files are available for the game in ../powerpit/slots:

1. test.txt, standard test configuration
- 5        2. slot.txt, standard game machine configuration; this is copied by bldslot.sh (see III.6. Building the CD-ROM) to ../slots/slot.txt on the target system.

The text of slot.txt follows:

```

10        // name of applet to run for game (required)
          GAME_APPLET        powerpit.slots.SlotMachine
          // dimensions for applet display (required)
          WIDTH              640
          HEIGHT             480
          // game denomination: cents per credit (required)
15        DENOMINATION        5
          // unreasonable win amount strictly greater than any possible jackpot
                              (credits)
          // defaults to 999,999,999
          WIN_LIMIT          1000000
20        // game controls when the error candle goes off and on - any value for this
          // parameter allows game-control of the error candle!
          MANUAL_CANDLE      true
          // tilt when player wins this many dollars in credits (clear with key)
          JACKPOT_LIMIT      10000
25        // turn the validators off when we have this many dollars in credits
          VALIDATORS_OFF     3000
          // handpay when credits are greater than this much in dollars
          HAND_PAY            1200
          // turn off coin validator when this many dollars are accepted between
30                            games
          COIN_LIMIT          1500
          // turn off bill validator when this many dollars are accepted between
                              games
          BILL_LIMIT          1500
35        // turn off eft transfers in when this many dollars are accepted between
                              games
          EFT_LIMIT           3000
          // turn the validators off when we have this many dollars in credits
          VALIDATORS_OFF     3000
40        // handpay when credits are greater than this much in dollars
          HAND_PAY            50
          // turn off coin validator when this many dollars are accepted between
                              games

```

```

COIN_LIMIT      1500
// turn off bill validator when this many dollars are accepted between
                    games
BILL_LIMIT      1500
5 // turn off eft transfers in when this many dollars are accepted between
                    games
EFT_LIMIT       3000
// station type (required)
STATION         powerpit.devices.PlayerStation
10 STATION_CLASSES powerpit.mains.BaseGameInterface
STATION_VALUES  owner
// dealer image to display in background (required for graphics)
PLAYER_BACKGROUND slots/princess/images/background.gif
// random number generator to use (required for dealer or standalone)
15 GENERATOR      powerpit.chance.KnuthWithRNG
GENERATOR_CLASSES java.lang.Integer
GENERATOR_VALUES  200
// dealer candle
CANDLE          powerpit.slots.NVSMCandles
20 CANDLE_CLASSES powerpit.devices.PlayerStation
CANDLE_VALUES   owner
// online system (comment this out to test bill validator)
ONLINE          powerpit.online.OnlineSystem
ONLINE_CLASSES  powerpit.devices.PlayerStation
25 ONLINE_VALUES owner
COMPANY         "IGCA "
PAYBACK         96.84
GAME_ID         "NVSSF "
// iiob
30 IIOB          powerpit.iiob.IIOB
IIOB_CLASSES    powerpit.devices.Station
IIOB_VALUES     owner
// meters
METER_MODEL     powerpit.slots.NVSMMeterModel
35 METER_MODEL_CLASSES powerpit.devices.Station
METER_MODEL_VALUES owner
// meter log for recovery
METER_LOG       powerpit.mains.nvram.MeterLog
METER_LOG_CLASSES java.lang.Integer
40 METER_LOG_VALUES  5
// event log for alarms (last 100)
EVENT_LOG       powerpit.mains.nvram.EventLog
EVENT_LOG_CLASSES java.lang.Integer java.lang.Integer
EVENT_LOG_VALUES  5 100
45 // state log for recovery
STATE_LOG       powerpit.mains.nvram.StateLog
STATE_LOG_CLASSES powerpit.mains.BaseGameInterface
                    java.lang.Integer

```

```

STATE_LOG_VALUES      owner 5
// history log for play steps (last 50 = last 25 games)
PLAY_LOG              powerpit.mains.nvram.PlayLog
PLAY_LOG_CLASSES      java.lang.Integer java.lang.Integer
5  PLAY_LOG_VALUES     5 50
// bill log
BILL_LOG              powerpit.mains.nvram.BillLog
BILL_LOG_CLASSES      java.lang.Integer java.lang.Integer
10 BILL_LOG_VALUES     5 25
// switches
SWITCH_HARNESS        powerpit.slots.NVSMSwitchHarness
SWITCH_HARNESS_CLASSES powerpit.devices.Station
SWITCH_HARNESS_VALUES owner
// button panel
15 BUTTONS             powerpit.slots.NVSMButtonPanel
BUTTONS_CLASSES        powerpit.devices.PlayerStation
                        java.lang.Boolean
BUTTONS_VALUES         owner false
// bill validator
20 BILL_VALIDATOR       powerpit.devices.JCMValidator
BILL_VALIDATOR_CLASSES powerpit.devices.PlayerStation
                        java.lang.Integer
BILL_VALIDATOR_VALUES  owner 0
NOTE: the JCM bill validator may also be tested by connecting it directly
25 to the serial port of your PC. This is done by replacing the last 2 lines
above with:
BILL_VALIDATOR_CLASSES powerpit.devices.PlayerStation
                        java.lang.String
BILL_VALIDATOR_VALUES  owner "/dev/cua0"
30 NOTE: use cua0 for MS COM1, cua1 for MS COM2
// coin validator
COIN_VALIDATOR         powerpit.devices.ColeCoinValidator
COIN_VALIDATOR_CLASSES powerpit.devices.PlayerStation
                        java.lang.Integer java.lang.Integer java.lang.Integer
35                        java.lang.Integer java.lang.Integer
// coin-in, tilt, drop gate, hopper full, power
COIN_VALIDATOR_VALUES  owner 5 24 6 9 4
// hopper
HOPPER                 powerpit.devices.Hopper
40 HOPPER_CLASSES        powerpit.devices.CoinOutMeter
                        java.lang.Integer java.lang.Integer
// motor line, sensor line
HOPPER_VALUES          owner 7 8
// printer
45 //PRINTER             powerpit.devices.Printer
//PRINTER_CLASSES      powerpit.devices.PlayerStation
                        java.lang.Integer
//PRINTER_VALUES       owner 1

```

```

// reel symbols parameter order is: (enclose in quotes if they contain
//      spaces)
// owner
5 // reel symbol name as it appears in the reel strips (all upper case)
// reel symbol display names (singular form)
// reel symbol display names (plural form)
// reel symbol image filename
// reel strip symbol name of alternate symbol for win flashing
10 REEL_SYMBOLS      powerpit.slots.ReelSymbol
REEL_SYMBOLS_CLASSES powerpit.slots.ReelSymbols
      java.lang.String
      java.lang.String
      java.lang.String java.lang.String java.lang.String
15 REEL_SYMBOLS_VALUES owner CASTLE  Castle  Castles
      slots/princess/images/castle.gif BLANK
      owner PRINCESS Princess  Princesses
      slots/princess/images/princess.gif BLANK
      owner WITCH  Wizard  Wizards
20      slots/princess/images/wizard1.gif BLANK
      owner FAIRY  Fairy  Fairies
      slots/princess/images/fairy.gif BLANK
      owner KING  King  Kings
      slots/princess/images/king.gif BLANK
25      owner QUEEN  Queen  Queens
      slots/princess/images/queen.gif BLANK
      owner JACK  Jack  Jacks
      slots/princess/images/jack.gif BLANK
      owner TEN  10  10s
30      slots/princess/images/10.gif BLANK
      owner FROGPC Frog  Frogs
      slots/princess/images/frog.gif BLANK
      owner SPELLBK "Spell Book" "Spell Books"
      slots/princess/images/books.gif BLANK
35      owner BLANK  Blank  Blanks
      slots/princess/images/blank.gif BLANK

//reel strips
REEL_STRIPS      powerpit.slots.princess.ReelStrips96
REEL_STRIPS_CLASSES powerpit.slots.SlotMachine
40 REEL_STRIPS_VALUES owner
//combination table
COMBINATION_TABLE      powerpit.slots.princess.PrincessCombo
COMBINATION_TABLE_CLASSES powerpit.slots.SlotMachine
COMBINATION_TABLE_VALUES owner
45 // upper left corner of first reel
REEL_X      70
REEL_Y      138
// pixels between reels

```

REEL\_SPACE 12

```

//meter digits (and space character)
ONSCREEN_METER_DIGITS slots/princess/images/Dig 11
5 //onscreen meter coordinates
ONSCREEN_METER_CREDIT_X 60
ONSCREEN_METER_CREDIT_Y 24
ONSCREEN_METER_BET_X 177
ONSCREEN_METER_BET_Y 24
10 ONSCREEN_METER_WIN_X 274
ONSCREEN_METER_WIN_Y 24
ONSCREEN_METER_PAID_X 383
ONSCREEN_METER_PAID_Y 24
PAID_PATCH_X 342
15 PAID_PATCH_Y 0
PAID_PATCH slots/princess/images/paid.gif
//paylines in their "turned on" state
PAYLINES slots/princess/images/Line 9
//paylines in their "extra bright" state
20 //PAYLINES_BRIGHT slots/princess/images/Gloline 9
//// sound to play for big payoff
//WIN_SOUND slots/audio/irealshort8.wav
// sounds to play for wins
WIN_TUNES powerpit.slots.WinTune
25 WIN_TUNES_CLASSES powerpit.slots.WinTunes java.lang.Integer
java.lang.String
WIN_TUNES_VALUES owner 30 slots/audio/irealshort8.wav
owner 60 slots/audio/irealshort18.wav
owner 200 slots/audio/lshort8.wav
30 owner 750 slots/audio/medium8.wav
// sound to play while reels are spinning
INDEX_SOUND slots/audio/our_reel_stop.wav
The game specific module comprising specific game engine may be
located in ../powerpit/slots and ../powerpit/slots/princess.
35 slots/SlotMachine.java: main video slot machine engine
slots/NVSMButtonPanel.java: container for assigning buttons.
slots/NVSMCandles.java: container for assigning candles.
slots/NVSMMeterModel.java: container for assigning meters.
slots/NVSMSwitchHarness.java: container for assigning switches.
40 Utility classes for organizing reel symbols and virtual reel-strips:
slots/ReelPositions.java
slots/ReelStrips.java
slots/ReelSymbol.java
slots/ReelSymbols.java
45 slots/princess/ReelStrips90.java
slots/princess/ReelStrips92.java
slots/princess/ReelStrips94.java
slots/princess/ReelStrips96.java

```

Utility classes for computing combinations:  
 slots/SlotCombo.java  
 slots/SlotComboLine.java  
 slots/SubstitutionLine.java  
 5 State data and play history:  
 slots/SlotFinal.java  
 slots/SlotInitial.java  
 slots/SlotState.java  
 slots/SlotStateData.java  
 10 Analyzer:  
 slots/SlotTest.java  
 Utility classes for computing wins:  
 slots/SlotWin.java  
 slots/SlotWinLine.java  
 15 slots/SubstitutionLine.java  
 Utility classes for displaying wins:  
 slots/WinFlash.java  
 slots/WinFlashThread.java  
 slots/WinTune.java  
 20 slots/WinTunes.java

All files required to give the Monster Money Video Slot "personality" to the basic system described previously, reside in ../powerpit/slots/monster and directories, ../powerpit/slots/princess/ and ../powerpit/slots/audio/. Sounds data modules are stored  
 25 in ../powerpit/slots/audio/ and graphics for the reels and backgrounds are in ../powerpit/slots/monster/images/. NOTE: the only difference between Monster Money and Storybook Fantasy (for identical percentage pay tables) is the graphics and configuration file - all executable code is identical.

Several configuration files are available for the game in  
 30 ../powerpit/slots/monster:  
 1. test.txt, standard test configuration  
 2. slot.txt, standard game machine configuration; this is copied by bldslot.sh to ../slots/slot.txt on the target system.

3. testtest.txt, test configuration with analyzer

35 The text of slot.txt follows:

```
// name of applet to run for game (required)
GAME_APPLET      powerpit.slots.SlotMachine
// dimensions for applet display (required)
WIDTH             640
```

HEIGHT 480

```

// game denomination: cents per credit (required)
DENOMINATION      5
5 // unreasonable win amount strictly greater than any possible jackpot (credits)
// defaults to 999,999,999
WIN_LIMIT         1000000
// game controls when the error candle goes off and on - any value for this
// parameter allows game-control of the error candle!
10 MANUAL_CANDLE   true
// tilt when player wins this many dollars in credits (clear with key)
JACKPOT_LIMIT     10000
// turn the validators off when we have this many dollars in credits
VALIDATORS_OFF    3000
15 // handpay when credits are greater than this much in dollars
HAND_PAY          1200
// turn off coin validator when this many dollars are accepted between games
COIN_LIMIT        1500
// turn off bill validator when this many dollars are accepted between games
20 BILL_LIMIT     1500
// turn off eft transfers in when this many dollars are accepted between games
EFT_LIMIT         3000
// turn the validators off when we have this many dollars in credits
VALIDATORS_OFF    3000
25 // handpay when credits are greater than this much in dollars
HAND_PAY          50
// turn off coin validator when this many dollars are accepted between games
COIN_LIMIT        1500
// turn off bill validator when this many dollars are accepted between games
30 BILL_LIMIT     1500
// turn off eft transfers in when this many dollars are accepted between games
EFT_LIMIT         3000
// station type (required)
STATION           powerpit.devices.PlayerStation
35 STATION_CLASSES powerpit.mains.BaseGameInterface
STATION_VALUES    owner
// dealer image to display in background (required for graphics)
PLAYER_BACKGROUND slots/princess/images/background.gif
// random number generator to use (required for dealer or standalone)
40 GENERATOR       powerpit.chance.KnuthWithRNG
GENERATOR_CLASSES java.lang.Integer
GENERATOR_VALUES  200
// dealer candle
CANDLE           powerpit.slots.NVSMCandles
45 CANDLE_CLASSES  powerpit.devices.PlayerStation
CANDLE_VALUES    owner
// online system (comment this out to test bill validator)
ONLINE           powerpit.online.OnlineSystem

```

```

ONLINE_CLASSES      powerpit.devices.PlayerStation
ONLINE_VALUES      owner
COMPANY            "IGCA "
PAYBACK            96.84
5  GAME_ID          "NVSSF "
// iioB
IIOB                powerpit.iioB.IIOB
IIOB_CLASSES        powerpit.devices.Station
IIOB_VALUES         owner
10 // meters
METER_MODEL         powerpit.slots.NVSMMeterModel
METER_MODEL_CLASSES powerpit.devices.Station
METER_MODEL_VALUES  owner
// meter log for recovery
15 METER_LOG         powerpit.mains.nvram.MeterLog
METER_LOG_CLASSES   java.lang.Integer
METER_LOG_VALUES    5
// event log for alarms (last 100)
EVENT_LOG           powerpit.mains.nvram.EventLog
20 EVENT_LOG_CLASSES java.lang.Integer java.lang.Integer
EVENT_LOG_VALUES    5 100

// state log for recovery
STATE_LOG           powerpit.mains.nvram.StateLog
25 STATE_LOG_CLASSES powerpit.mains.BaseGameInterface
    java.lang.Integer
STATE_LOG_VALUES    owner 5
// history log for play steps (last 50 = last 25 games)
PLAY_LOG           powerpit.mains.nvram.PlayLog
30 PLAY_LOG_CLASSES  java.lang.Integer java.lang.Integer
PLAY_LOG_VALUES    5 50
// bill log
BILL_LOG           powerpit.mains.nvram.BillLog
35 BILL_LOG_CLASSES  java.lang.Integer java.lang.Integer
BILL_LOG_VALUES    5 25
// switches
SWITCH_HARNESS     powerpit.slots.NVSMSwitchHarness
SWITCH_HARNESS_CLASSES powerpit.devices.Station
SWITCH_HARNESS_VALUES owner
40 // button panel
BUTTONS            powerpit.slots.NVSMButtonPanel
BUTTONS_CLASSES     powerpit.devices.PlayerStation java.lang.Boolean
BUTTONS_VALUES      owner false
// bill validator
45 BILL_VALIDATOR    powerpit.devices.JCMValidator
BILL_VALIDATOR_CLASSES powerpit.devices.PlayerStation
    java.lang.Integer
BILL_VALIDATOR_VALUES owner 0

```

NOTE: the JCM bill validator may also be tested by connecting it directly to the serial port of your PC. This is done by replacing the last 2 lines above with:

```

5  BILL_VALIDATOR_CLASSES powerpit.devices.PlayerStation java.lang.String
    BILL_VALIDATOR_VALUES owner "/dev/cua0"
    NOTE: use cua0 for MS COM1, cua1 for MS COM2
    // coin validator
    COIN_VALIDATOR      powerpit.devices.ColeCoinValidator
    COIN_VALIDATOR_CLASSES powerpit.devices.PlayerStation
10         java.lang.Integer java.lang.Integer java.lang.Integer
            java.lang.Integer java.lang.Integer
    // coin-in, tilt, drop gate, hopper full, power
    COIN_VALIDATOR_VALUES owner 5 24 6 9 4
    // hopper
15  HOPPER      powerpit.devices.Hopper
    HOPPER_CLASSES      powerpit.devices.CoinOutMeter
            java.lang.Integer java.lang.Integer
    // motor line, sensor line
    HOPPER_VALUES      owner 7 8
20  // printer
    //PRINTER      powerpit.devices.Printer
    //PRINTER_CLASSES      powerpit.devices.PlayerStation java.lang.Integer
    //PRINTER_VALUES      owner 1
    // reel symbols parameter order is: (enclose in quotes if they contain spaces)
25  // owner
    // reel symbol name as it appears in the reel strips (all upper case)
    // reel symbol display names (singular form)
    // reel symbol display names (plural form)
    // reel symbol image filename
30  // reel strip symbol name of alternate symbol for win flashing
    REEL_SYMBOLS      powerpit.slots.ReelSymbol
    REEL_SYMBOLS_CLASSES powerpit.slots.ReelSymbols java.lang.String
            java.lang.String
            java.lang.String java.lang.String java.lang.String
35  REEL_SYMBOLS_VALUES owner CASTLE "Monster money"
    "Monster monies"
        slots/monster/images/M_M3.gif BLANK
        owner PRINCESS Dragon Dragons
        slots/monster/images/Dragon3.gif BLANK
40  owner WITCH Crown Crowns
        slots/monster/images/Crown2.gif BLANK
        owner FAIRY Castle Castles
        slots/monster/images/Castle3.gif BLANK
        owner KING Coin Coins
45  slots/monster/images/Coins3.gif BLANK
        owner QUEEN Flame Flames
        slots/monster/images/flames3.gif BLANK
        owner JACK Knight Knights

```

```

        slots/monster/images/Knight3.gif  BLANK
owner TEN  Shield  Shields
        slots/monster/images/Shield3.gif  BLANK
owner FROGPCR Eyes  Eyes
5        slots/monster/images/eyes1.gif  BLANK
owner SPELLBK Jewels Jewels
        slots/monster/images/Jewels3.gif  BLANK
owner BLANK  Blank  Blanks
        slots/monster/images/blank.gif  BLANK
10      //reel strips
        REEL_STRIPS      powerpit.slots.princess.ReelStrips96
        REEL_STRIPS_CLASSES powerpit.slots.SlotMachine
        REEL_STRIPS_VALUES owner
        //combination table
15      COMBINATION_TABLE      powerpit.slots.princess.PrincessCombo
        COMBINATION_TABLE_CLASSES powerpit.slots.SlotMachine
        COMBINATION_TABLE_VALUES owner

        // upper left corner of first reel
20      REEL_X      70
        REEL_Y      138
        // pixels between reels
        REEL_SPACE      12
        //meter digits (and space character)
25      ONSCREEN_METER_DIGITS slots/monster/images/0.gif
        slots/monster/images/1.gif
        slots/monster/images/2.gif
        slots/monster/images/3.gif
        slots/monster/images/4.gif
30      slots/monster/images/5.gif
        slots/monster/images/6.gif
        slots/monster/images/7.gif
        slots/monster/images/8.gif
        slots/monster/images/9.gif
35      slots/monster/images/10.gif

        //onscreen meter coordinates
        ONSCREEN_METER_CREDIT_X 60
        ONSCREEN_METER_CREDIT_Y 26
        ONSCREEN_METER_BET_X 200
40      ONSCREEN_METER_BET_Y 26
        ONSCREEN_METER_WIN_X 312
        ONSCREEN_METER_WIN_Y 26
        ONSCREEN_METER_PAID_X 434
        ONSCREEN_METER_PAID_Y 26
45      PAID_PATCH_X      385
        PAID_PATCH_Y      0
        PAID_PATCH      slots/monster/images/PAID.gif
        //paylines in their "turned on" state

```

```

PAYLINES          slots/monster/images/newLine 9
//// sound to play for big payoff
//WIN_SOUND       slots/audio/irealshort8.wav
// sounds to play for wins
5  WIN_TUNES       powerpit.slots.WinTune
   WIN_TUNES_CLASSES powerpit.slots.WinTunes java.lang.Integer
                           java.lang.String
   WIN_TUNES_VALUES owner 30  slots/audio/irealshort8.wav
                           owner 60  slots/audio/irealshort18.wav
10  owner 200  slots/audio/Ishort8.wav
   owner 750  slots/audio/imedium8.wav

// sound to play while reels are spinning
INDEX_SOUND       slots/audio/our_reel_stop.wav
15  The game specific module comprising specific game engine may be located
   in.../powerpit/slots and .../powerpit/slots/princess.
   slots/SlotMachine.java: main video slot machine engine
   slots/NVSMButtonPanel.java: container for assigning buttons.
   slots/NVSMCandles.java: container for assigning candles.
20  slots/NVSMMeterModel.java: container for assigning meters.
   slots/NVSMSwitchHarness.java: container for assigning switches.
   Utility classes for organizing reel symbols and virtual reel-strips:
   slots/ReelPositions.java
   slots/ReelStrips.java
25  slots/ReelSymbol.java
   slots/ReelSymbols.java
   slots/princess/ReelStrips90.java
   slots/princess/ReelStrips92.java
   slots/princess/ReelStrips94.java
30  slots/princess/ReelStrips96.java
   Utility classes for computing combinations:
   slots/SlotCombo.java
   slots/SlotComboLine.java
   slots/SubstitutionLine.java
35  State data and play history:
   slots/SlotFinal.java
   slots/SlotInitial.java
   slots/SlotState.java
   slots/SlotStateData.java
40  Analyzer:
   slots/SlotTest.java
   Utility classes for computing wins:
   slots/SlotWin.java
   slots/SlotWinLine.java
45  slots/SubstitutionLine.java
   Utility classes for displaying wins:
   slots/WinFlash.java
   slots/WinFlashThread.java

```

slots/WinTune.java  
 slots/WinTunes.java

5 All files required to give the multi-player, Nevada-style Blackjack "personality" to  
 the basic system described previously, reside in ../powerpit/blackjack and its  
 subdirectories, audio/ and images/. Sounds data modules are stored in audio/ and  
 graphics for the cards and backgrounds are in images/.

The class powerpit.blackjack.BJDealer is a subclass of  
 powerpit.blackjack.BJBase, which defines data structures and a few methods common to  
 10 the player and dealer. BJBase is a subclass of GameApplet.

Several configuration files are available for the dealer:

1. lvd1.txt, standard test configuration

2. gamed1.txt, standard game machine configuration; this is copied by  
 nvbj\_copy.sh to ../blackjack/lvd1.txt on the target system.

15 gamed1.txt follows:

```

// lvd1.txt: run player station with remote dealer
// name of applet to run for game (required)
GAME_APPLET      powerpit.blackjack.BJDealer
// dimensions for applet display (required)(adjust for size error)
20 WIDTH          660
   HEIGHT         520
// game denomination: cents per credit (required)
DENOMINATION      100
// station type (required)
25 STATION         powerpit.devices.DealerStation
   STATION_CLASSES powerpit.mains.BaseGameInterface
   STATION_VALUES  owner
// player stations to load (required for dealer)
PLAYERS           5
30 // dealer image to display in background (required for graphics)
   DEALER_BACKGROUND blackjack/images/Bjdback.gif
// random number generator to use (required for dealer or standalone)
GENERATOR         powerpit.chance.JKnuthV2Random
GENERATOR_CLASSES java.lang.Integer
35 GENERATOR_VALUES 200
// dealer candle
CANDLE           powerpit.blackjack.NVBJDCandles
CANDLE_CLASSES   powerpit.devices.DealerStation
CANDLE_VALUES    owner
40 // iio
IIOB             powerpit.iio.IIOB
IIOB_CLASSES     powerpit.devices.Station
```

```

IIOB_VALUES          owner

// meters
METER_MODEL          powerpit.blackjack.NVBJDMeterModel
5 METER_MODEL_CLASSES powerpit.devices.Station
METER_MODEL_VALUES   owner
// meter log for recovery
METER_LOG            powerpit.mains.nvram.MeterLog
METER_LOG_CLASSES    java.lang.Integer
10 METER_LOG_VALUES   5
// event log for alarms
EVENT_LOG            powerpit.mains.nvram.EventLog
EVENT_LOG_CLASSES    java.lang.Integer java.lang.Integer
EVENT_LOG_VALUES     5 100
15 // state log for recovery
STATE_LOG            powerpit.mains.nvram.StateLog
STATE_LOG_CLASSES    powerpit.mains.BaseGameInterface java.lang.Integer
STATE_LOG_VALUES     owner 5
// switches
20 SWITCH_HARNESS     powerpit.blackjack.NVBJDSwitchHarness
SWITCH_HARNESS_CLASSES powerpit.devices.Station
SWITCH_HARNESS_VALUES owner
// card image location
DECK                 blackjack/images/
25 // maximum bet allowed
MAX_BET              50
// sound to play if the dealer busts out
BUST_SOUND           blackjack/audio/dbust.au
// sounds to play for hand values
30 NUM_SOUNDS         blackjack/audio/
// sound to play at end of hand
END_SOUND            blackjack/audio/spacemusic.au
// sound to play for accepting bets
PLACEBETS_SOUND      blackjack/audio/dpls.au
35 // sound to play for no more bets
NOMOREBETS_SOUND     blackjack/audio/dnmb.au
// sound to play just before dealing
LUCK_SOUND           blackjack/audio/dglk.au
// sound to play when idling
40 LETSPLAY_SOUND     blackjack/audio/dlpbj.au
// sound to play for insurance opportunity
INSURE_SOUND         blackjack/audio/dins.au

The game specific module comprising specific game engine may be located
45 in.../powerpit/blackjack/BJDealer.java.
The class powerpit.blackjack.BJPlayer is a subclass of powerpit.blackjack.BJBase,
which defines data structures and a few methods common to the player and
dealer. BJBase is a subclass of GameApplet.

```

Several configuration files are available for the players:

1. lvp1.txt, standard test configuration
2. gamep1.txt, standard game machine configuration; this is copied by nvbj\_copy.sh to ../blackjack/lvp1.txt on the target system.

```

5  gamep1.txt follows:
   // lvp1.txt: run player station with remote dealer
   // name of applet to run for game
   GAME_APPLET      powerpit.blackjack.BJPlayer
   // maximum bet
10  MAX_BET          50
   // dimensions for applet display (required)(adjust for size error)
   WIDTH             660
   HEIGHT            520
   // game denomination: cents per credit (required)
15  DENOMINATION     100
   // station type (required)
   STATION            powerpit.devices.NetworkPlayerStation
   STATION_CLASSES    powerpit.mains.BaseGameInterface
   STATION_VALUES     owner
20  // player image to display in background
   PLAYER_BACKGROUND blackjack/images/Bjpback.gif
   // player lamp for attendant
   CANDLE             powerpit.blackjack.NVBJPCandles
   CANDLE_CLASSES     powerpit.devices.PlayerStation
25  CANDLE_VALUES    owner
   // iiob
   IIOB               powerpit.iiob.IIOB
   IIOB_CLASSES       powerpit.devices.Station
   IIOB_VALUES        owner
30  // meters
   METER_MODEL        powerpit.blackjack.NVBJPMeterModel
   METER_MODEL_CLASSES powerpit.devices.Station
   METER_MODEL_VALUES owner
   // meter log for recovery
35  METER_LOG         powerpit.mains.nvram.MeterLog
   METER_LOG_CLASSES  java.lang.Integer
   METER_LOG_VALUES   5
   // event log for alarms
   EVENT_LOG          powerpit.mains.nvram.EventLog
40  EVENT_LOG_CLASSES java.lang.Integer java.lang.Integer
   EVENT_LOG_VALUES   5 100
   // history log for play steps
   PLAY_LOG           powerpit.mains.nvram.PlayLog
   PLAY_LOG_CLASSES   java.lang.Integer java.lang.Integer
45  PLAY_LOG_VALUES   5 25
   // switches
   SWITCH_HARNESS     powerpit.blackjack.NVBJPSSwitchHarness
   SWITCH_HARNESS_CLASSES powerpit.devices.Station

```

```

SWITCH_HARNESS_VALUES owner
// button panel
BUTTONS powerpit.blackjack.NVBJPButtonPanel
BUTTONS_CLASSES powerpit.devices.PlayerStation
5  BUTTONS_VALUES owner
// bill validator
BILL_VALIDATOR powerpit.devices.CBVBillValidator
BILL_VALIDATOR_CLASSES powerpit.devices.PlayerStation java.lang.Integer
BILL_VALIDATOR_VALUES owner 0
10 // coin validator
COIN_VALIDATOR powerpit.devices.BJCoinValidator
COIN_VALIDATOR_CLASSES powerpit.devices.PlayerStation
java.lang.Integer java.lang.Integer java.lang.Integer
java.lang.Integer java.lang.Integer java.lang.Integer
15 // coin-in, tilt, hopper gate, drop gate, hopper full, power
COIN_VALIDATOR_VALUES owner 5 24 0 6 9 4
// hopper
HOPPER powerpit.devices.Hopper
HOPPER_CLASSES powerpit.devices.CoinOutMeter
20 java.lang.Integer java.lang.Integer
// motor line, sensor line
HOPPER_VALUES owner 7 8

// card image location
25 DECK blackjack/images/
// sound to play if the player busts out
BUST_SOUND blackjack/audio/dbust.au
// sounds to play for hand values
NUM_SOUNDS blackjack/audio/
30 // sound to play at end of hand
END_SOUND blackjack/audio/spacemusic.au
// sound to play for winning hand
WIN_SOUND blackjack/audio/bfan.au
// sound to play for player blackjack
35 BJ_SOUND blackjack/audio/dbj.au
// sound to play for losing hand
//LOSE_SOUND blackjack/audio/hammer.au
// sound to play for push
40 PUSH_SOUND blackjack/audio/gong.au

```

The game specific module comprising specific game engine may be located in.../powerpit/blackjack/BJPlayer.java.

In addition to BJDealer, BJPlayer, the configuration files and sound and video files, the Nevada Blackjack game requires some specialized classes. This will provide a

good estimate of the total new code required for most games. Single player games (a simple slot example may be found in.../powerpit/docs) will require less additional code.

BJStateData: subclasses powerpit.mains.GameData to provide specialized state data for blackjack; uses BJHand and BJPlayerHand.

5 NVBJDCandles: subclasses powerpit.devices.Candelabra to provide specific functionality for the dealer candles on the blackjack machine.

BJFinal: subclasses powerpit.messages.FinalStep to provide final game outcome information specific to blackjack; uses BJHand and BJPlayerHand.

10 NVBJDMeterModel: subclasses powerpit.devices.MeterModel to provide specialized dealer meters.

BJHand: subclasses powerpit.chance.CardHand to provide specialized blackjack functionality.

NVBJDSwitchHarness: subclasses powerpit.devices.SwitchHarness to provide specific functionality for the dealer switches on the blackjack machine.

15 BJInitial: subclasses powerpit.messages.InitialStep to provide game start information specific to blackjack; uses BJHand.

NVBJPButtonPanel: subclasses powerpit.devices.ButtonPanel to provide normal player button functionality for the blackjack player station.

20 BJIntermediate: subclasses powerpit.messages.IntermediateStep to provide intermediate information specific to blackjack; uses BJHand and BJPlayerHand.

NVBJPCandles: subclasses powerpit.devices.Candelabra to provide specific functionality for the player lamps on the blackjack machine.

NVBJPMeterModel: subclasses powerpit.devices.MeterModel to provide specialized player meters.

25 BJPlayerHand: provides a container for up to 4 instances of BJHand and provides methods to determine whether a specific hand is splittable, etc.

NVBJPShowButtonPanel: subclasses powerpit.devices.ButtonPanel to provide show player button functionality for the blackjack player station.

30 BJState: subclasses powerpit.mains.GameState to provide values for additional states, e.g. INSURANCE, unique to blackjack.

NVBJPSwitchHarness: subclasses powerpit.devices.SwitchHarness to provide specific functionality for the player switches on the blackjack machine.

The foregoing description of the exemplary embodiment of the invention has been presented for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. It is intended that the scope of  
5 the invention be limited not with this detailed description, but rather by the claims appended hereto.

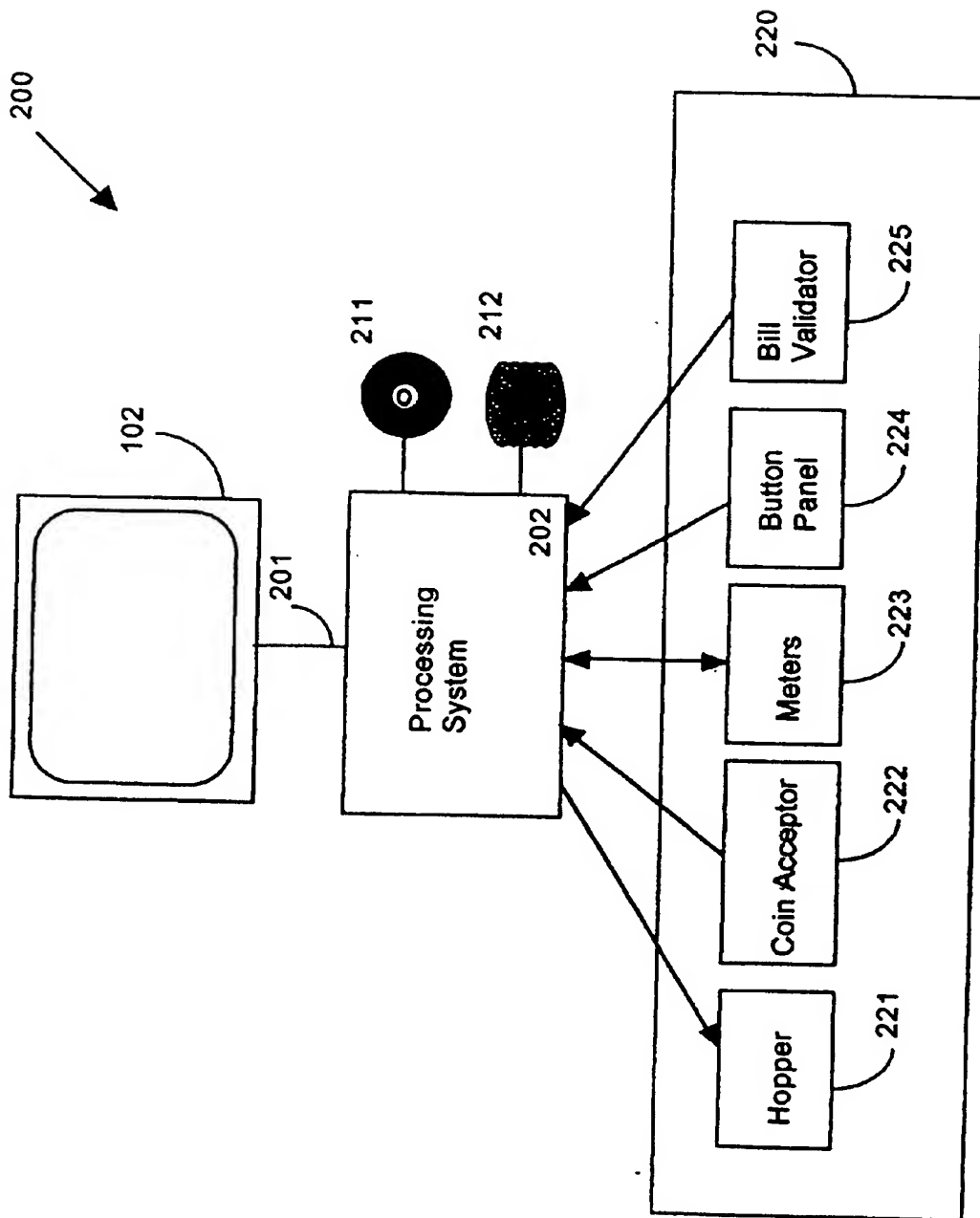
## WHAT IS CLAIMED IS:

1. An apparatus for compartmentalizing various game instruction modules within an overall architecture of a video gaming machine having a video display, an audio output module, and a plurality of user input buttons to provide re-usable instruction modules common to all games implemented on a video gaming platform, the apparatus comprising:
  - an open-source operating system;
  - a plurality of platform support and utility modules common to all games run on the video gaming machine, the plurality of platform support and utility modules comprise:
    - a wagering funds module for accepting and paying out funds associated with wagers played on the outcome of the games run on the video gaming machine;
    - a user interaction module for generating output display information and accepting user input instructions;
    - a multimedia output module for generating output signals on the video display and audio output module; and
    - a random number generator module for creating the random numbers used to generate the outcome of the games run on the video gaming machine; and
    - a plurality of game specific modules and associated data files, the plurality of game specific modules and associated data files comprise:
      - a game specific engine module for executing the instructions implementing the specific game on the video gaming machine;
      - a plurality of video image data files combined by the game specific modules to create a sequence of image displayed on the video display; and
      - a plurality of audio sound data files played by the game specific modules to create a series of sounds generated on the audio output module in conjunction with the sequence of image displayed on the video display.
2. The apparatus according to claim 1, wherein the wager funds module comprises:
  - a hopper module for returning winnings from wagers on the outcome of the games run on the video gaming machine; and

a bill and coin acceptance module for receiving funds for use in placing from wagers on the outcome of the games run on the video gaming machine.

3. The apparatus according to claim 1, wherein the user interaction module for  
5 generating output display information and accepting user input instructions comprises:  
a meter display and register module for maintaining and displaying a plurality of game parameter values associated with wagers, winnings, user funds balance, and the outcome of the games run on the video gaming machine; and  
a buttons module for accepting input signals associated with user  
10 instructions for playing wagers and generating the outcome of the games run on the video gaming machine.
4. The apparatus according to claim 1, wherein the multimedia output module for generating output signals on the video display and audio output module comprises:  
15 a audio output module for generating output audio signals on the audio output module; and  
a video output module for generating a sequence of output video images displayed upon the video display.
- 20 5. The apparatus according to claim 1, wherein the outcome of the games run on the video gaming machine is based upon a video slot machine game.
6. The apparatus according to claim 1, wherein the outcome of the games run on the video gaming machine is based upon a video blackjack game.
- 25 7. The apparatus according to claim 6, wherein the video blackjack game is a multi-player blackjack game comprising a plurality of video gaming devices networked into a multi-processor-based distributed gaming system.

FIG. 2



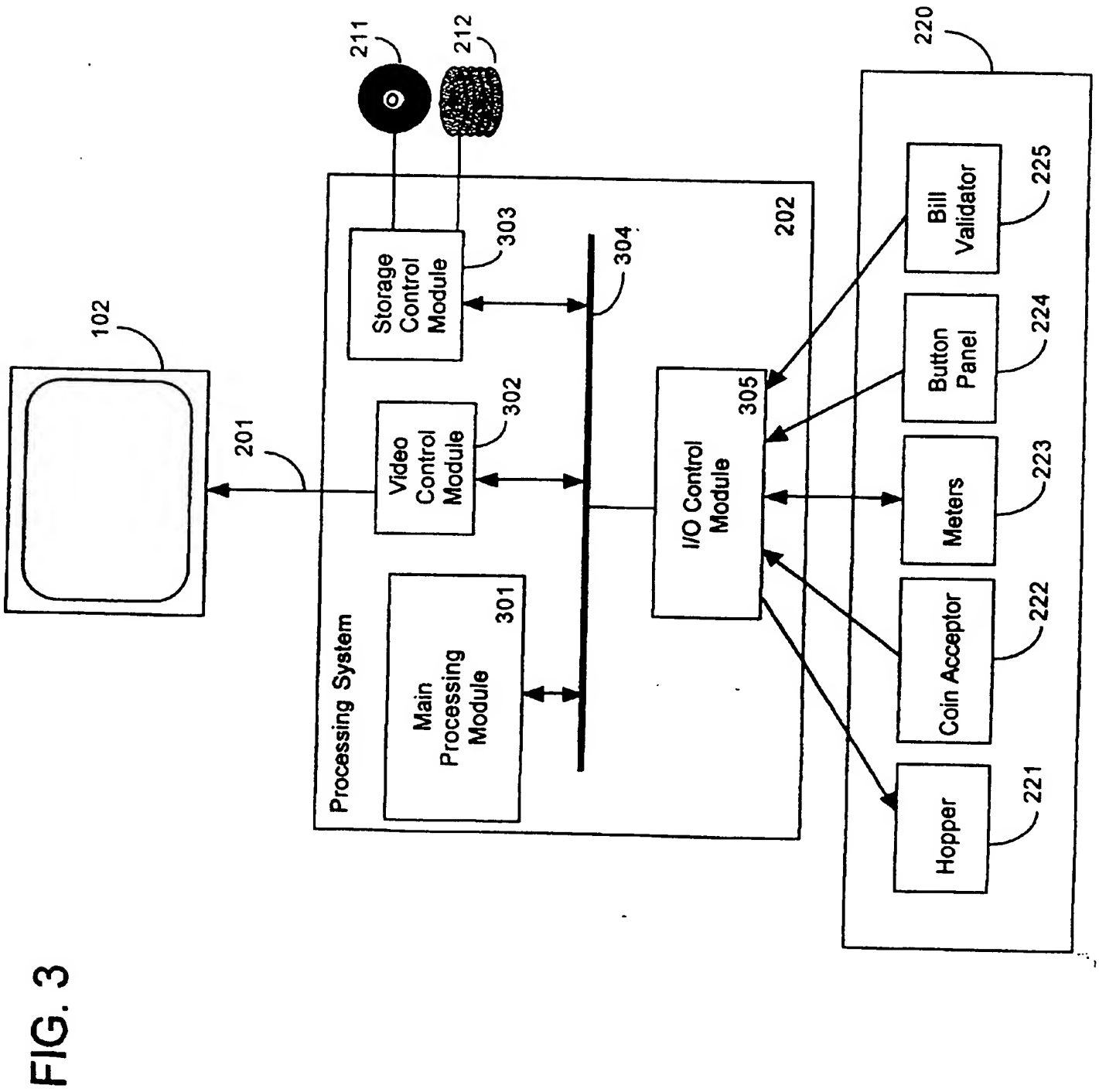


FIG. 4

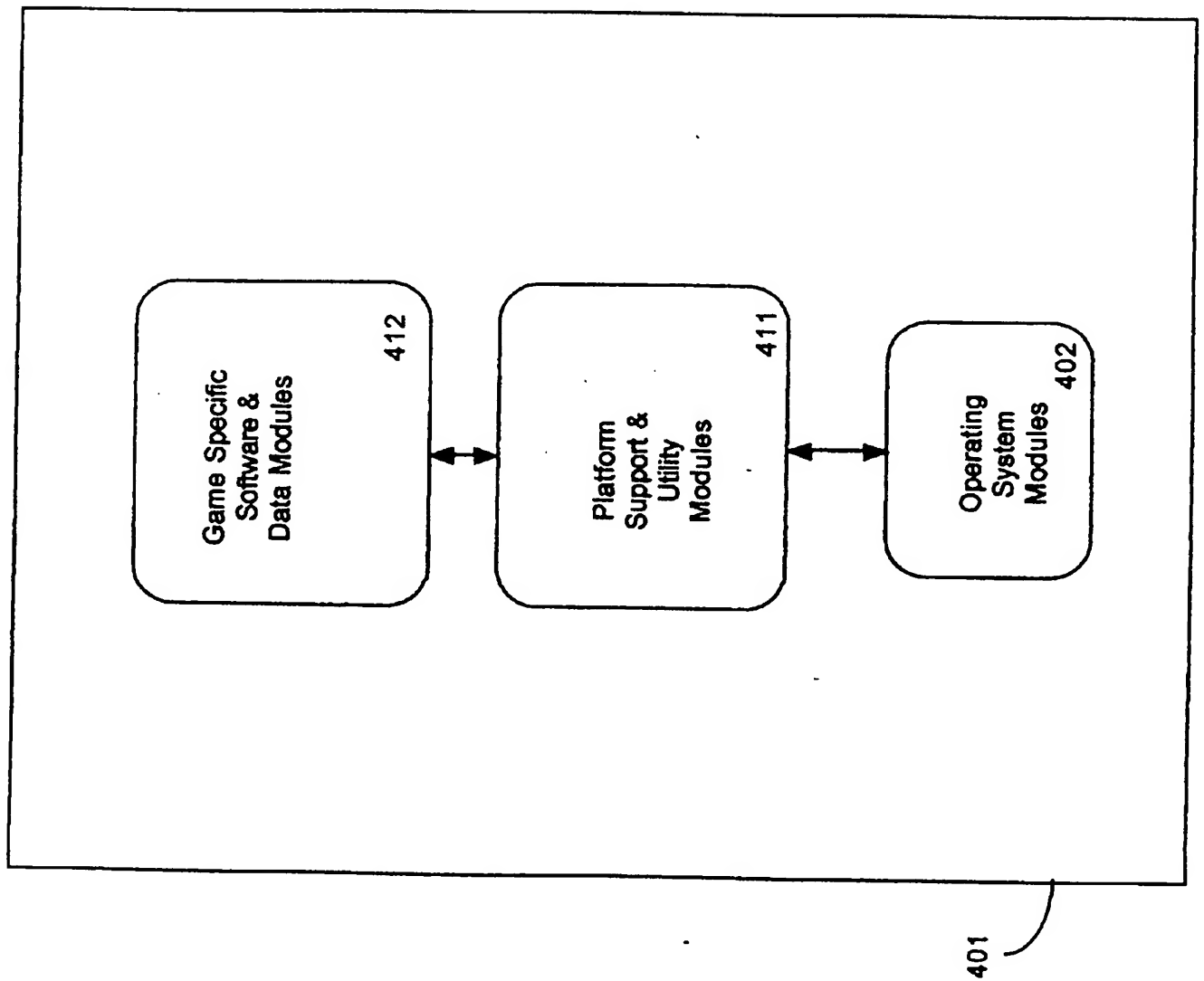
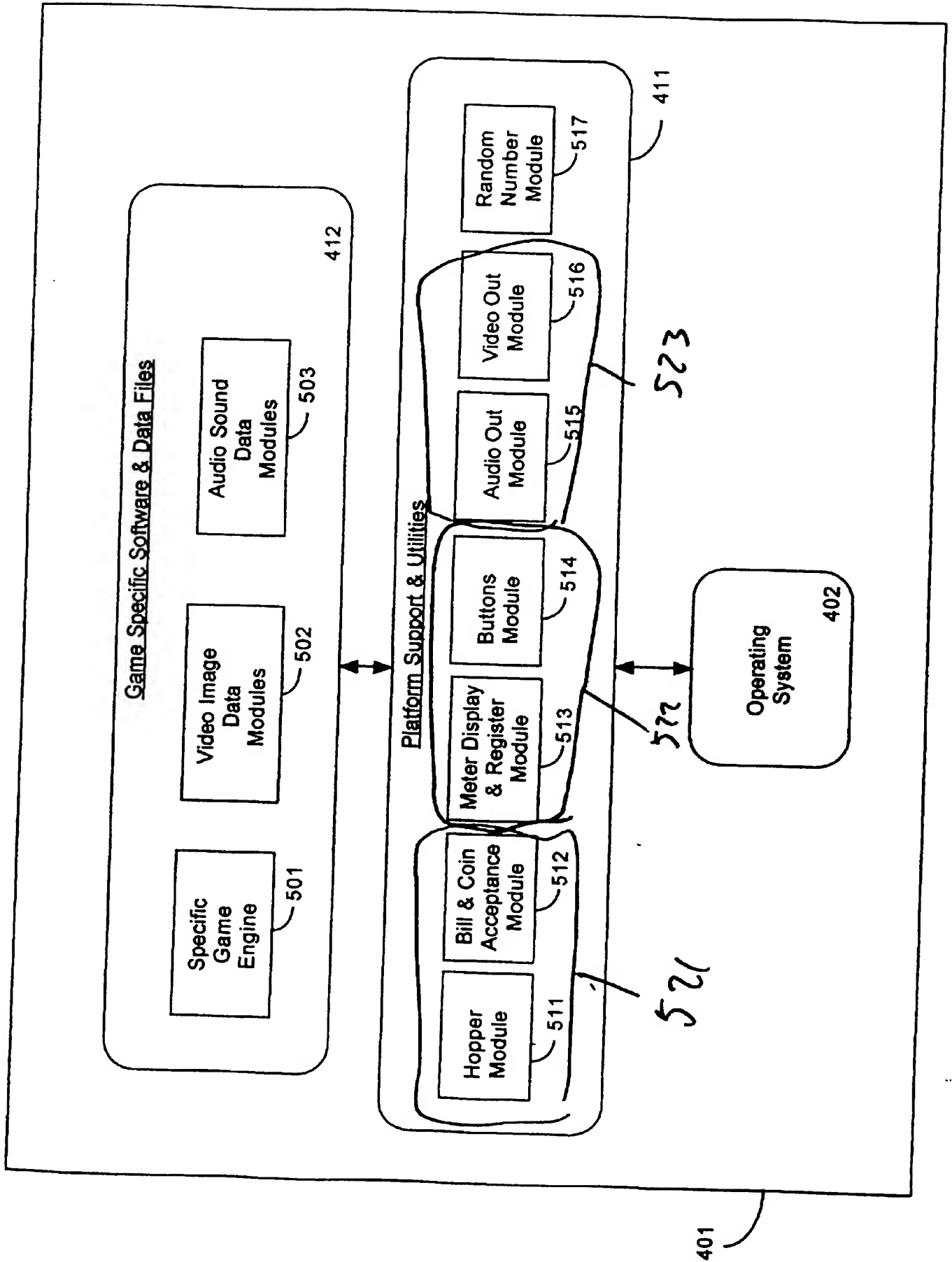


FIG. 5



## Game Application Context

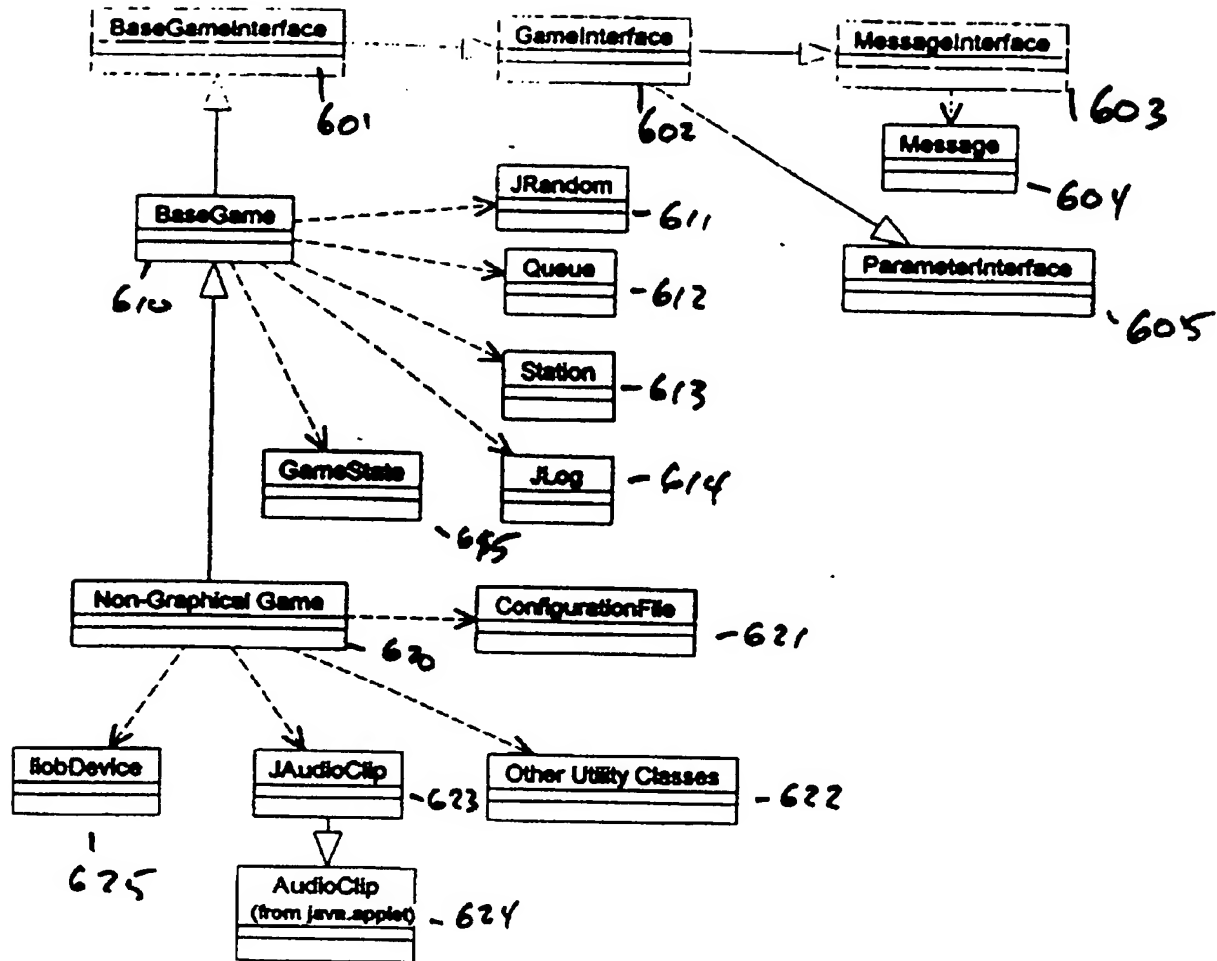


Fig 6a

## GameApplet Context

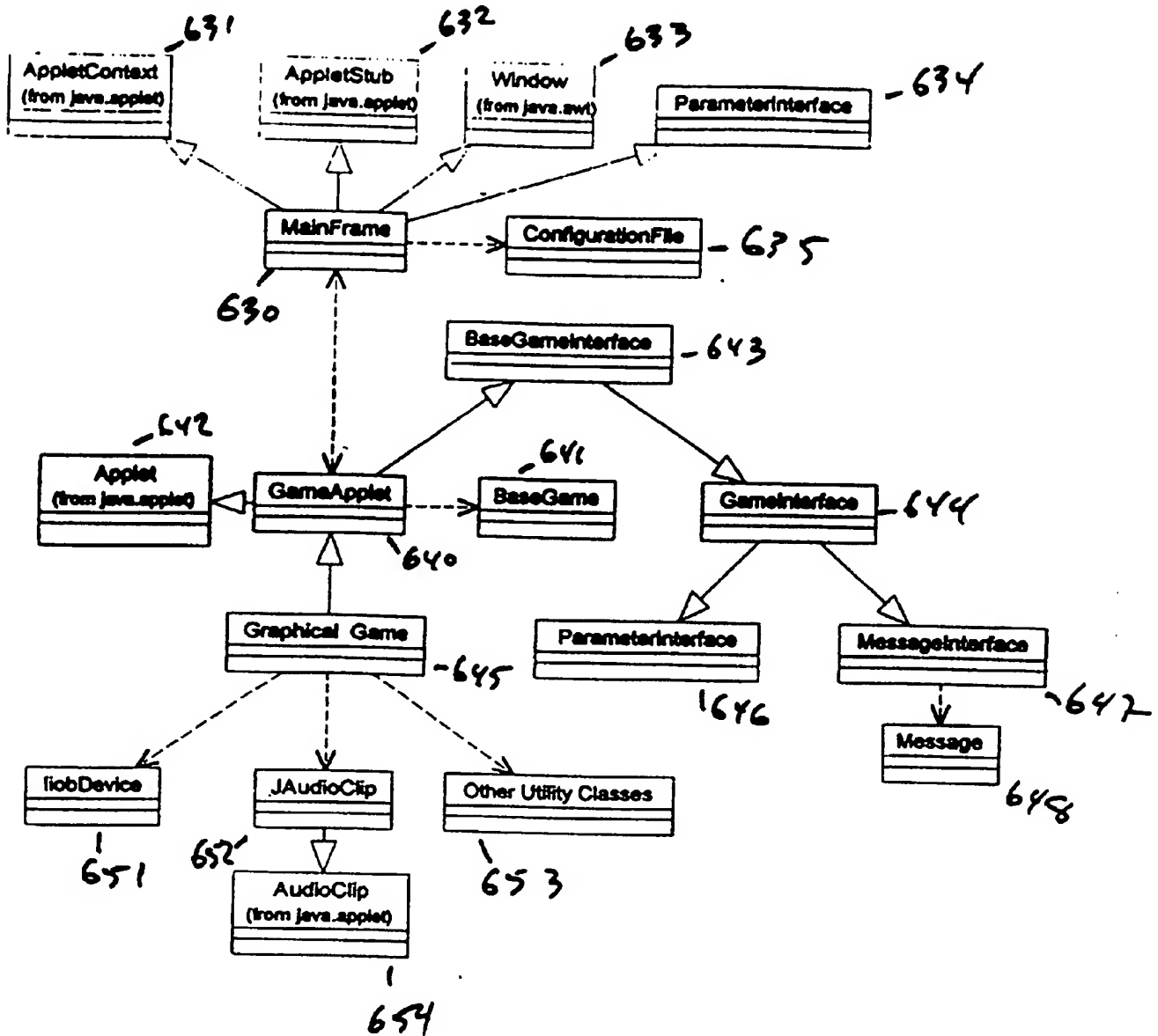


Fig 6B

## Station Context

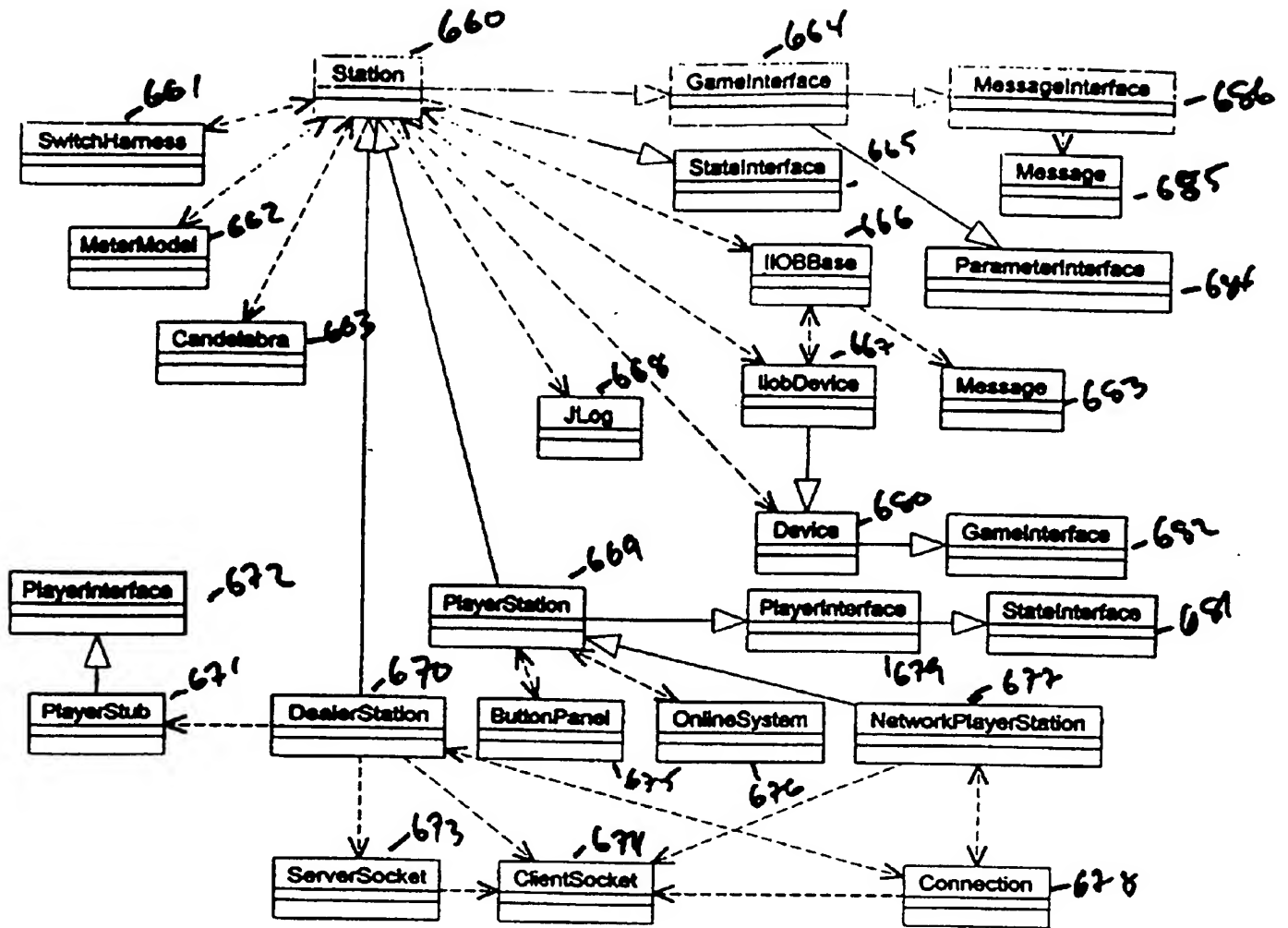


Fig 6c

## Device Layer

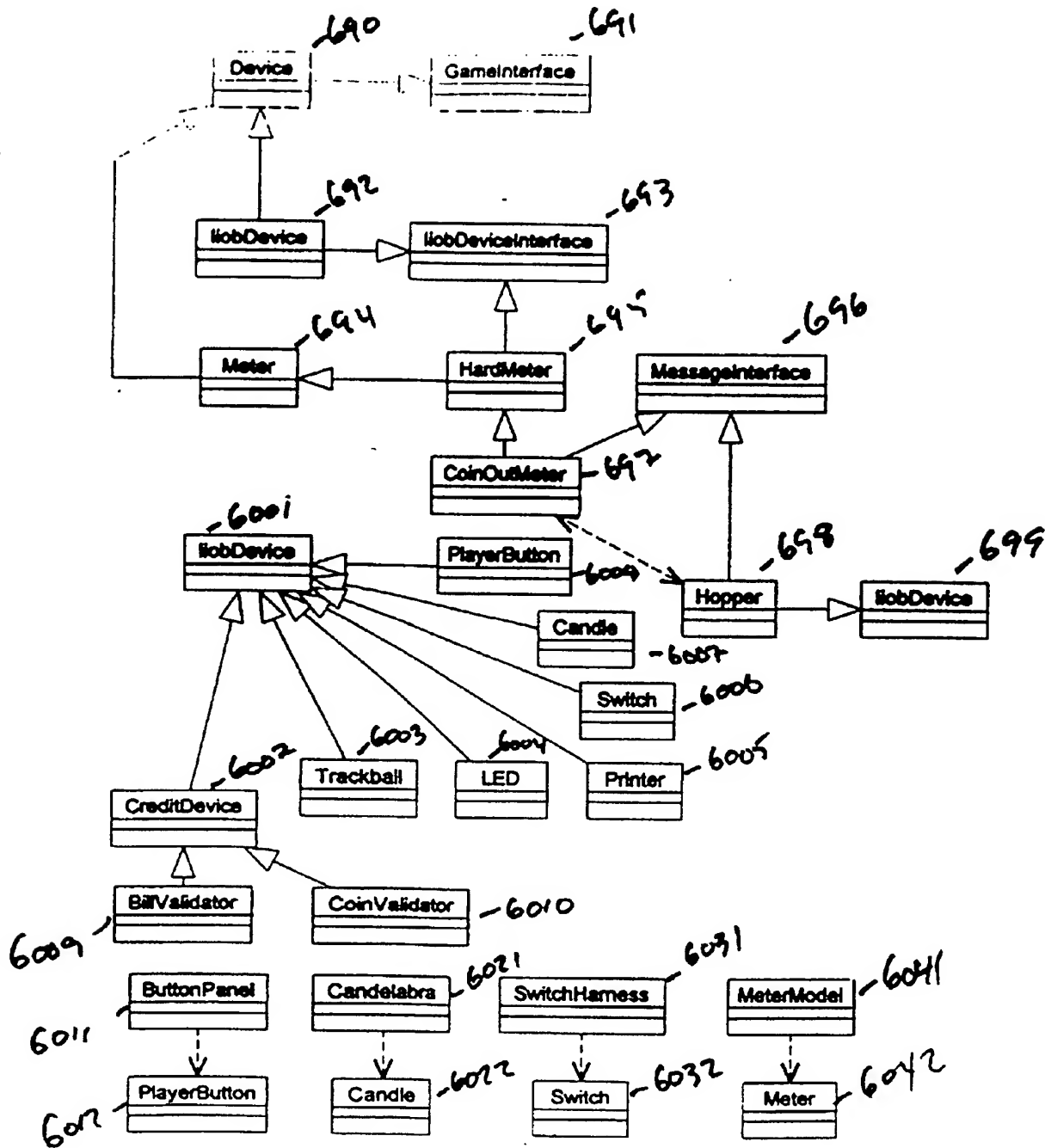


Fig 6d

## Multi-player Classes

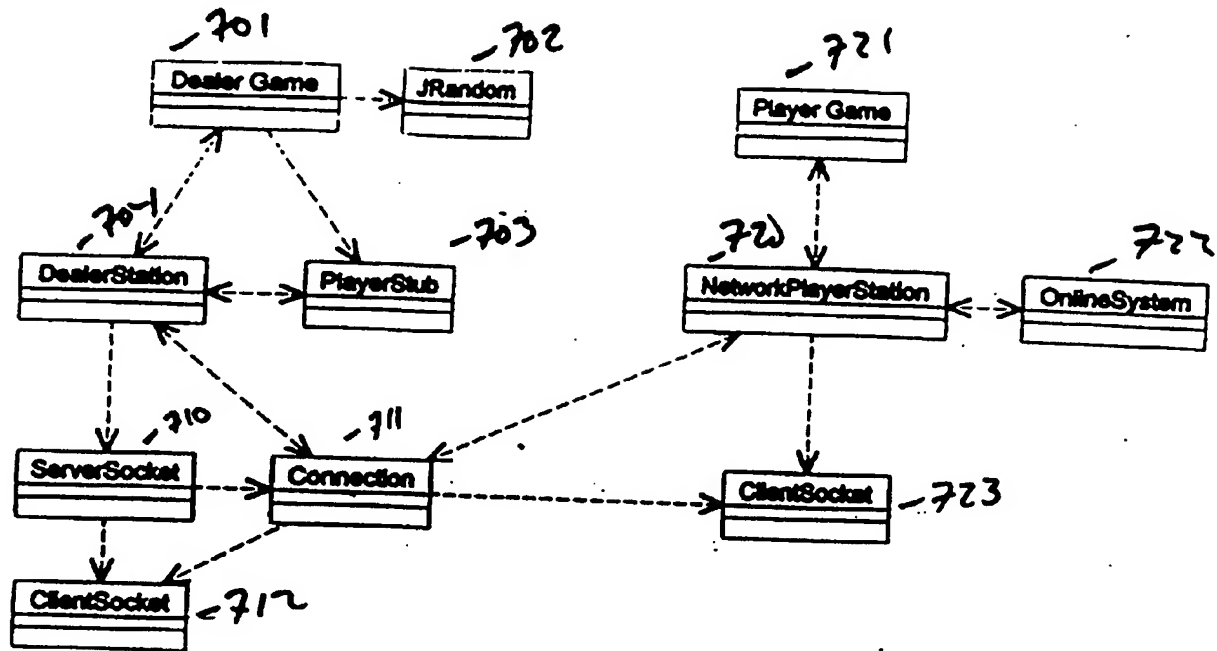
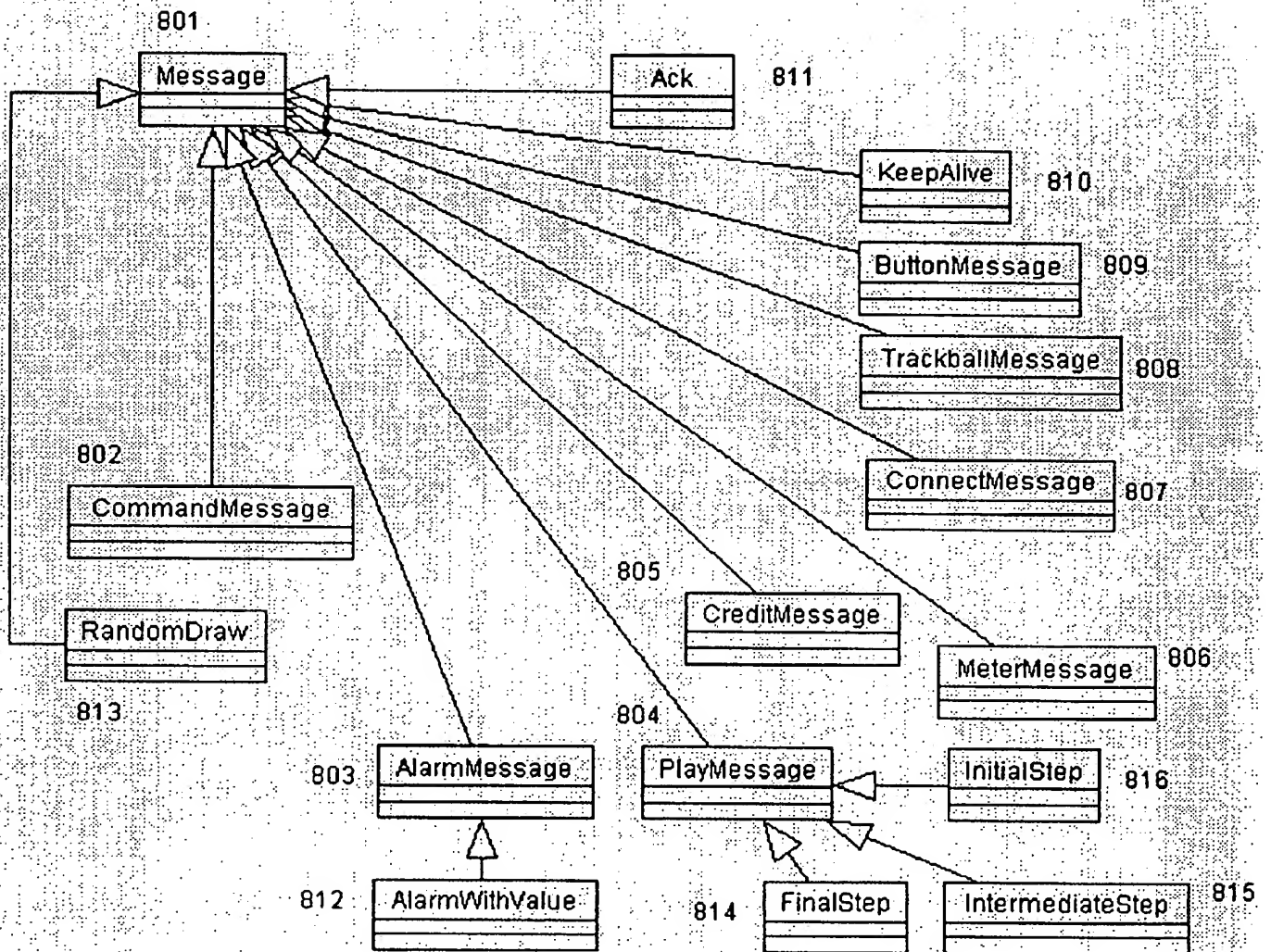
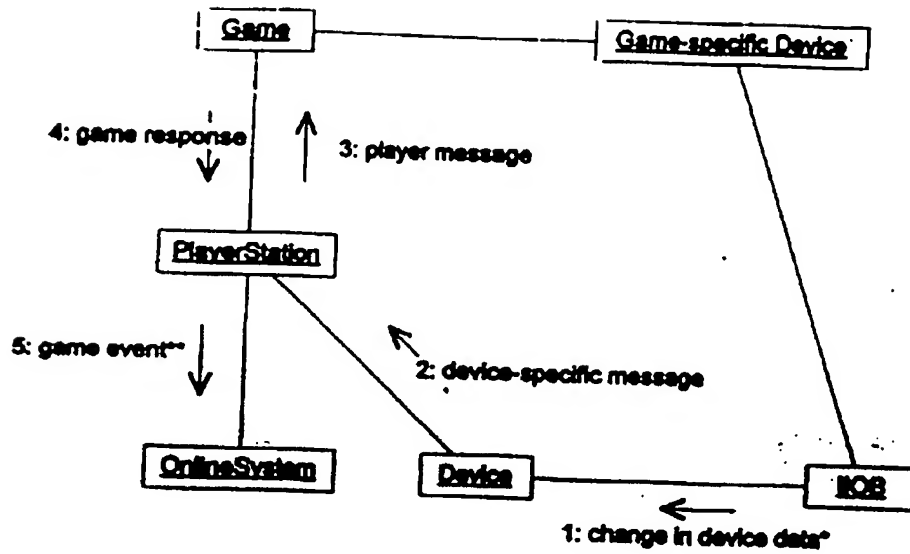


Fig. 7

Message Classes Fig 8.



## Local Message Passing

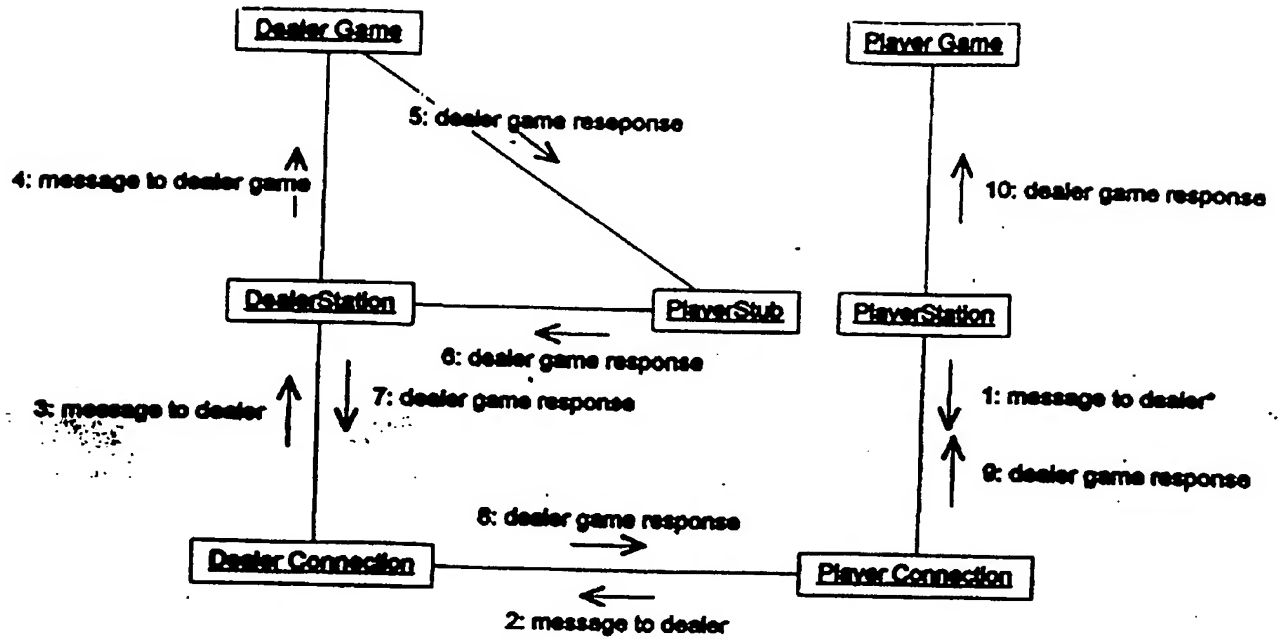


\*Note that a change in device data for a game-specific device (e.g.: slot-reel stepper motor, rotating rings device) would follow a path through the game-specific device, to the game, then to the player station.

\*\*Note that in some cases, the online system may generate game events (e.g. shutdown, eft credits or debits). This would be treated like a change in device data.

FIG. 9

# Network Message Passing



\*Note that the original message may come either from the game as a response to a locally generated event (e.g.: the end of a sound or animation sequence) or directly from the player station.

Fig-10

2319600

UNSCANNABLE ITEM

RECEIVED WITH THIS APPLICATION

(ITEM ON THE 10TH FLOOR ZONE 5 IN THE FILE PREPARATION SECTION)

DOCUMENT REÇU AVEC CETTE DEMANDE

NE POUVANT ÊTRE BALAYÉ

(DOCUMENT AU 10 IÈME ÉTAGE AIRE 5 DANS LA SECTION DE LA  
PRÉPARATION DES DOSSIERS)